Växjö University

School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

VizzAnalyzer goes Eclipse!

David Ruiz De Azua

Jun
2007

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 07064
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--07064/--SE

Master Thesis

VizzAnalyzer goes Eclipse!

David Ruiz de Azua Nieto

2nd of February 2007

Department of Computer Science
Växjö University

Supervisors:
Rüdiger Lincke, Phil. Lic.

Abstract

The VizzAnalyzer Framework is a stand-alone tool for analyzing and visualizing the structures of large software systems. Today, it has its own limited Swing based GUI lacking a professional look & feel. Furthermore, the effort needed to extend the VizzAnalyzer with new features like automatic update, progress monitoring, help system, and integration of the Eclipse Java and C/C++ AST API is high.

In order to improve current limitations and ease the future maintenance effort we refactored the VizzAnalyzer to be a plug-in to the Eclipse platform. We removed the burden of GUI development from the authors of the VizzAnalyzer replacing the Swing GUI with a SWT based GUI, which utilizes the rich feature set provided by the Eclipse Platform. Furthermore, the we did not only provide existing features of the VizzAnalyzer as loading and binding graphs, a complex system to load dynamic plug-ins functionalities for analysis, retrieval and visualization. We implemented an update and help manager, allowed for an easy use of third party plug-ins, which are available for Eclipse, and provided product branding.

We propose that the newly created VizzAnalyzer 2.0 solved the aforementioned limitations and provides a good foundation for the future evolution of the VizzAnalyzer tool.

This master thesis documents our how the VizzAnalyzer 2.0 has been developed and implemented for the Eclipse platform, and how developers shall use the new VizzAnalyzer version.

Table of Contents

1 Introduction 1
1.1 Context of the thesis 1
1.2 Problem 1
1.3 Goals and Criteria 2
1.4 Motivation 2
1.5 Outline 3
2 Background 4
2.1 The VizzAnalyzer Framework 4
2.2 The AWT/Swing architecture 5
2.3 The SWT architecture 5
2.4 AWT/Swing versus SWT 6
2.4.1 Swing component hierarchy 6
2.4.2 SWT component hierarchy 7
2.4.3 Compare Swing versus SWT code 7
2.5 NetBeans versus Eclipse 10
3 Requirements 11
3.1 Users 11
3.2 Features 11
3.3 Use Cases 11
3.3.1 Use Case Model 12
3.3.2 Use Cases 13
3.4 Functional requirements 16
4 Architecture 19
4.1 The Eclipse architecture 19
4.1.1 The Eclipse Plug-in Model 20
4.1.2 Extension 20
4.2 The VizzAnalyzer architecture 21
5 Design and implementation 23
5.1 The VizzAnalyzer 1.0 24
5.2 The VizzAnalyzer 2.0 25
5.3 VizzAnalyzer extensions 29
5.4 VizzAnalyzer tree viewer 32
5.5 VizzAnalyzer views 35
5.6 Help manager 38
5.7 Update manager 39
5.8 Console 43
5.9 Loading dynamic plug-ins 45
6 Conclusions and Future Work 48
6.1 Conclusions 48
6.2 Future Work 50
References 52
Appendix A. The VizzAnalyzer 2.0 55
A.1 User Manual 55
A.1.1 Installation and running 55
A.1.2 Product key 55
A.1.3 Loading/saving graphs 57
A.1.4 Help manager 59
A.1.5 Update manager 61

A.2 Developer Manual 62
A.2.1 Updating help content 62
A.2.2 Update site manager 64
A.2.3 Exporting VizzAnalyzer 66
A.2.4 How to use the VizzAnalyzer console 68
A.2.5 How to use the progress service 68
A.2.6 Class path Configuration 70

List of Figures

Figure 2.1: The VizzAnalyzer Framework architecture....4
Figure 2.2: Overview about AWT/Swing layered architecture....5
Figure 2.3: SWT architecture....6
Figure 2.4: Swing component hierarchy....6
Figure 2.5: SWT component hierarchy....7
Figure 2.6: AWT/Swing sample code....8
Figure 2.7: SWT sample code....9
Figure 2.8: Hello World example programs....10
Figure 3.1: Use Case Model of VizzAnalyzer 2.0....12
Figure 4.1: Eclipse architecture, red parts are relevant for thesis....19
Figure 4.2: Plugin.xml....20
Figure 4.3: The VizzAnalyzer Framework's architecture....21
Figure 4.4: The VizzAnalyzer 2.0 architecture....22
Figure 5.1: GUI design from VizzAnalyzer 1.0....24
Figure 5.2: VizzAnalyzer plugin.xml....25
Figure 5.3: VizzAnalyzer Class diagram....26
Figure 5.4: Action Class Diagram....27
Figure 5.5: VizzAnalyzer Core Sequence Diagram....29
Figure 5.6: VizzAnalyzer Extensions....30
Figure 5.7: VizzAnalyzer plug-ins and fragments....31
Figure 5.8: Plug-in dependencies....32
Figure 5.9: Graph Model Class Diagram....32
Figure 5.10: Graph content provider....33
Figure 5.11: Graph View with tree viewer....34
Figure 5.12: GraphView tree viewer....35
Figure 5.13: Views Class diagram....35
Figure 5.14: Sample view implementation....37
Figure 5.15: Graph properties view result....37
Figure 5.16: List help manager dependencies....38
Figure 5.17: Updating action for help manager....38
Figure 5.18: List plug-ins for the update feature....39
Figure 5.19: Adding update site information....40
Figure 5.20: Updating action implementation....41
Figure 5.21: AddExtension action implementation....42
Figure 5.22: Manage Extension action implementation....43
Figure 5.23: Console implementation....45
Figure 5.24: Screenshot VizzAnalyzer console....45
Figure 5.25: VizzAnalyzer 1.0 loading dynamic plug-ins....46
Figure 5.26: VizzAnalyzer 2.0 loading dynamic plug-ins....47
Figure 6.1: VizzAnalyzer 1.0 GUI....49
Figure 6.2: VizzAnalyzer 2.0 GUI....50
Figure A.1: Installation folder....55
Figure A.2: Registration dialog....56
Figure A.3: Default perspective....56
Figure A.4: Loading graphs menu....57
Figure A.5: Loading graphs dialog....57
Figure A.6: Graphs loaded....58
Figure A.7: Tabs information....58

Figure A.8: Perspectives 59
Figure A.9: Help manager menu 59
Figure A.10: Help manager content 60
Figure A.11: Help manager query 60
Figure A.12: Update manager menu 61
Figure A.13: Update progress 61
Figure A.14: Checking updates 61
Figure A.15: Taking new version 62
Figure A.16: Updating help content 63
Figure A.17: toc.xml 63
Figure A.18:toctasks.xml 63
Figure A.19: Update site created 64
Figure A.20: Site editor 65
Figure A.21: URL site 66
Figure A.22: Configuring delta pack plug-ins 66
Figure A.23: Exporting to other platforms 67
Figure A.24: Select platforms to export 67
Figure A.25: VizzAnalyzer OS 68
Figure A.26: How to use the console 68
Figure A.27: Progress service busy indicator sample 69
Figure A.28: Progress service job sample 70
Figure A.29: Plug-ins required 71
Figure A.30: Sharing JARs 71
Figure A.31: VizzAnalyzer plug-ins required 71
Figure A.32: Visualization Plug-ins share 71

Glossary

API	Application Programming Interface
JDT	Java Development Tools
JRE	Java Runtime Environment
OOP	Object Oriented Programming
PDE	Plug-in Development Environment (PDE)
RCP	Rich Client Platform
SWT	Standard Widgets Toolkit
GUI	Graphic User Interface

1 Introduction

The Java platform is extensively used on server and enterprise applications. Java is even used on systems that require high performance, like scientific applications, or games. Despite its extensive use on server platforms, it has not been used too much on desktop applications. The main reason for this is the look & feel and performance of Java applications with a graphical user interface, which is different and slower, as other graphical user interfaces, an end-user of applications is accustomed to. Until the latest Java version, there have been improvements from Sun Microsystems, but a full integration providing a platform dependent look & feel, in particular speed, is still missing.

To improve on that, IBM developed as part of the Eclipse project the Standard Widgets Toolkit (SWT), which is a class library for creating graphical user interfaces (GUIs). SWT makes it possible to create Java-based applications that are indistinguishable from a platform's native applications by means of look & feel and GUI related performance.

Furthermore, the Eclipse Project, which started as a Java IDE, allows through its extensible framework to be used for enterprise development, embedded and device development, rich client platform (RCP), application frameworks and language IDE for the most popular languages (C/C++, PHP, COBOL, and others). It is nowadays one of the important development environments and platforms for developers and application development.

Thus, since Eclipse provides an open, extensible framework, which allows to independently develop tools that integrate with third party tools seamlessly. Eclipse tools can interact between each other so seamless, that it is sometimes not possible to know when one tool ends and another starts.

Its main value receives Eclipse from the fact, that it is a platform, which is very flexible and extensible, allowing reducing the development effort of Graphical User Interfaces significantly by reusing its framework, components, or plug-ins. To assure consistency in the user interface between the registered components of the platform, standards and guidelines have been defined (see [7]). Thanks to these guidelines and the flexible architecture of the platform consistency within the platform and between integrated tools can be assured. The notation of tool disappears completely thanks to the cohesion in Eclipse Platform.

1.1 Context of the thesis

The VizzAnalyzer Framework is a stand-alone tool for analyzing and visualizing large software systems' structures [1]. Currently it has its own limited Swing based GUI. To remove the burden of GUI development, the authors of the VizzAnalyzer plan to replace the Swing GUI until the next VizzAnalyzer version with the Eclipse platform, by integrating the VizzAnalyzer as extension (plug-in) into Eclipse taking advantage of the rich feature set of this platform [2].

1.2 Problem

The object for this thesis is a complicated refactoring and reengineering task, which is porting the VizzAnalyzer framework currently controlled over its own Swing based GUI to the Eclipse Platform. The Swing based GUI components shall be replaced with elements of the Eclipse Platform using SWT, and then controlling the VizzAnalyzer program functions. The created Eclipse extension shall offer the same functionality as the current VizzAnalyzer GUI.

This task is difficult to solve, since the VizzAnalyzer framework is complex, a separation between GUI and program functions is not well described and the architecture needs to be recovered from the source code. Furthermore, a plug-in framework extends the GUI during program start with menu entries. Additionally, the Eclipse Platform API is complex and the mapping of the functions from the VizzAnalyzer GUI to the Eclipse GUI needs to be well understood and designed.

Thus, the problem addressed by this thesis is:

"Port the VizzAnalyzer Framework from Swing architecture to SWT architecture into the Eclipse Platform offering the same functionality as the VizzAnalyzer version 1.0 GUI".

1.3 Goals and Criteria

This section describes the goals pursued by this thesis in order to solve the problem and the criteria used for validating the goals:

The first goal is to reverse engineer the VizzAnalyzer 1.0, in order to get a good understanding its architecture and implementation. The important parts of its architecture shall be documented. This goal is met when the old and new VizzAnalyzer architecture is documented, and it is possible to design a SWT-based architecture under Eclipse Platform.

The second goal is the design and implementation of an alternative VizzAnalyzer GUI allowing for a improved look & feel matching the host platform. This is fulfilled when the new VizzAnalyzer has a GUI having the same look & feel as the platform it is running on, thereby providing the same functionality as the original VizzAnalyzer. Basic functionalities as load/save graphs, display graph properties, edge and node information are essential. Scripting and recording menus needs to be provided as well as their basic functionality. Furthermore, a set of dynamic plug-ins must be loaded on the VizzAnalyzer startup. Analyzer, Visualization and Front-end menus shall be present after startup.

The third goal is to extend the new VizzAnalyzer using new features from Eclipse platform. This is fulfilled when a progress service is implemented for long operations; also automatic update and help system are utilized.

The fourth goal is the portability of the new VizzAnalyzer. This is fulfilled when the VizzAnalyzer is functional on Windows, Linux and UNIX, and the look & feel is the same as the OS where it is running.

The fifth and last goal is the improvement of the performance; this is fulfilled when the GUI related actions are performed faster, or when process information is presented for long activities, e.g., loading graphs.

1.4 Motivation

Eclipse Rich Client Platform (RCP) offers for applications a more professional look & feel and reduced development costs, since the provided framework delivers much functionality, which otherwise needs to be implemented from scratch. Thus we hope to reduce the development and maintenance effort, and to provide more professional Graphic User Interface end-users, thereby increasing the acceptance in the student and researcher community.

Additionally, based on the new platform product branding is supported. Furthermore, we want to use the Eclipse Java and C++ AST API to parse Java and C++ programs. This is currently only fully supported from within the Eclipse framework.

1.5 Outline

The structure of this paper thesis is as follows, Chapter 2 describes background about AWT/Swing and SWT, in particular Swing and SWT architectures are explained giving and overview about those technologies. Chapter 3 provides an overview about the features, use cases and functional requirements derived from the existing VizzAnalyzer version, which shall be respected and extended in the new version. Chapter 4 covers the Eclipse architecture as a base architecture for the new VizzAnalyzer 2.0. We explain the concepts evolving on the Eclipse Platform. Chapter 5 describes the design and implementation of the new VizzAnalyzer 2.0. It compares in its subchapters the old and new architecture thereby underlining the changes between the old and new implementation. Finally, Chapter 6 concludes the thesis and describes future work.

Appendix A contains an end-user manual and developer manual.

2 Background

This section describes a brief description of VizzAnalyzer Framework, AWT/Swing and SWT architectures which are referenced throughout this thesis. First we present a brief overview about each of the architectures, then, we compare them against each other highlighting advantages and disadvantages. Furthermore, we provide coding samples to provide a more vivid picture of how the differences between the two API's affect the writing of source code.

2.1 The VizzAnalyzer Framework

The VizzAnalyzer Framework is a reusable framework for a rapid composition of reverse engineering tools. It is extensible for new programming languages, analyses, and visualization tools [1].

Figure 2.1: The VizzAnalyzer Framework architecture depicts the main components. "The framework consists of the framework-core, frozen-spots and hot-spots. The framework-core is responsible for communicating information between the different reverse engineering components connected to the framework. It has the functionality of a controller and information converter. The frozen-spots are in-house and externally developed reusable components supporting the framework-core with main functionalities. For instance, configurations necessary for reverse engineering tool compositions reuse our tiny-xml editor, a tool reused by the framework-core. Hot-spots are technically realized as directories and allow the simple and fast connection of arbitrary reverse engineering components with the framework." [1].

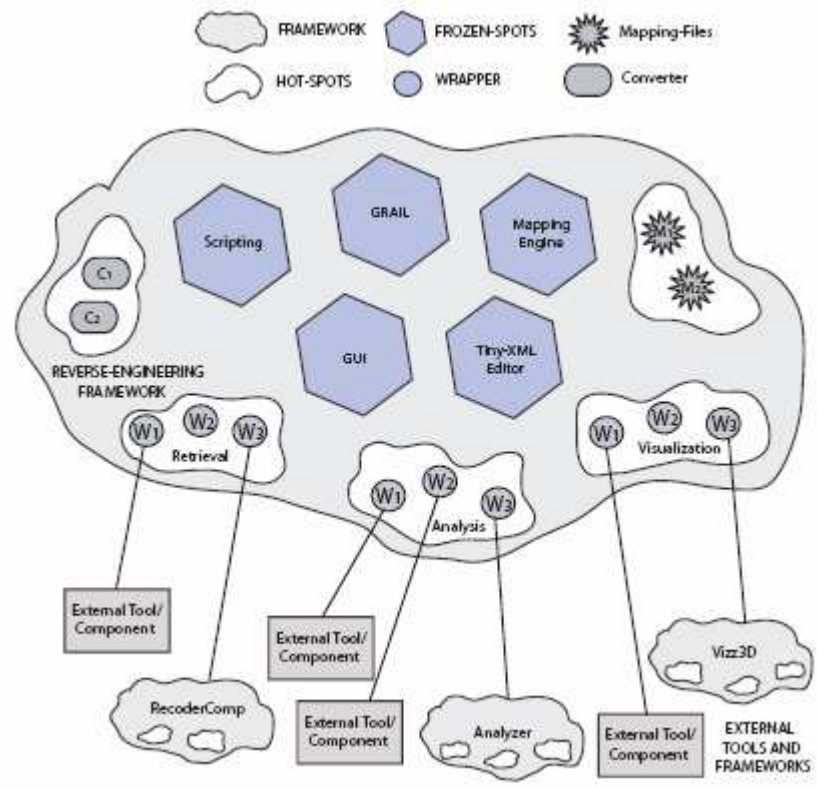

Figure 2.1: The VizzAnalyzer Framework architecture

For more information about the VizzAnalyzer refers too www.arisa.se or [1].

2.2 The AWT/Swing architecture

AWT/Swing is developed by Sun and part of the Java Runtime Environment. As seen in Figure 2.2: Overview about AWT/Swing layered architecture AWT and Swing have a layered architecture building on top of the operating system. The AWT library uses Java and C libraries to provide platform independence, so any computer running an operating system for which a Java Runtime Environment (JRE) is available can execute the software.

The user interface is independent of the operation system, therefore having always the same look & feel. There is no call to the windowing sub-system of the host operating system and that provides a level of platform independence.

The AWT library was not very flexible and some components were missing. Improving on that, Swing has been developed on the top of AWT, thereby providing more flexibility. Swing follows a Model View Controller (MVC) paradigm allowing for a flexible graphical user interface. Because of this architecture, performance has always been an issue, when using AWT and in particular Swing. Furthermore look & feel of the underlying operating system was not provided.

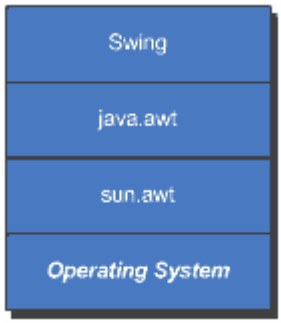

Figure 2.2: Overview about AWT/Swing layered architecture

2.3 The SWT architecture

SWT was initially developed by IBM and given to the open source community afterwards. As seen in Figure 2.3, SWT has also a layered architecture, but it communicates over JNI and some native SWT code directly with the Windowing Sub-system of the operating system on the host platform. Therefore SWT is faster compared with the Swing library, and the look & feel is the same as of the host platform.

SWT is a standalone library that can be used outside of any environment but it does not follow an Object Oriented Programming (OOP) when implementation is needed. JFace is developed on the top of SWT adding structure and facilities for common UI notions, and is designed to work with SWT [3]. The JFace library helps by adding many services to SWT applications. JFace does not hide SWT; it just extends it and one of its most important extensions is to isolate the application's data model from the GUI that displays and changes it. Yet, using SWT has some disadvantages compare with Swing. It is required the native SWT libraries on the target platform, and it is therefore not as flexible as Swing.

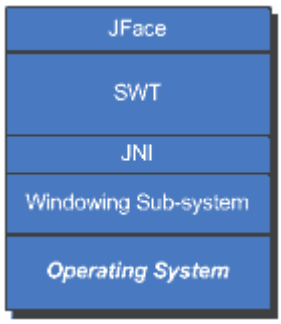

Figure 2.3: SWT architecture

2.4 AWT/Swing versus SWT

After we described briefly the overall AWT/Swing and SWT architecture in the previous sections, we are now going into some more detail, considering the component and inheritance hierarchy for the respective API's. We are closing this chapter with a small example comparing Swing and SWT code by showing how to build a simple application that contains a text, button and event listener.

2.4.1 Swing component hierarchy

The gray boxes are abstract classes and are not intended to be used directly. The yellow boxes are classes and can be deployed in a user interface. All Swing components inherit from JComponent as a base class. Container class is a generic AWT object containing other AWT components and JComponents.

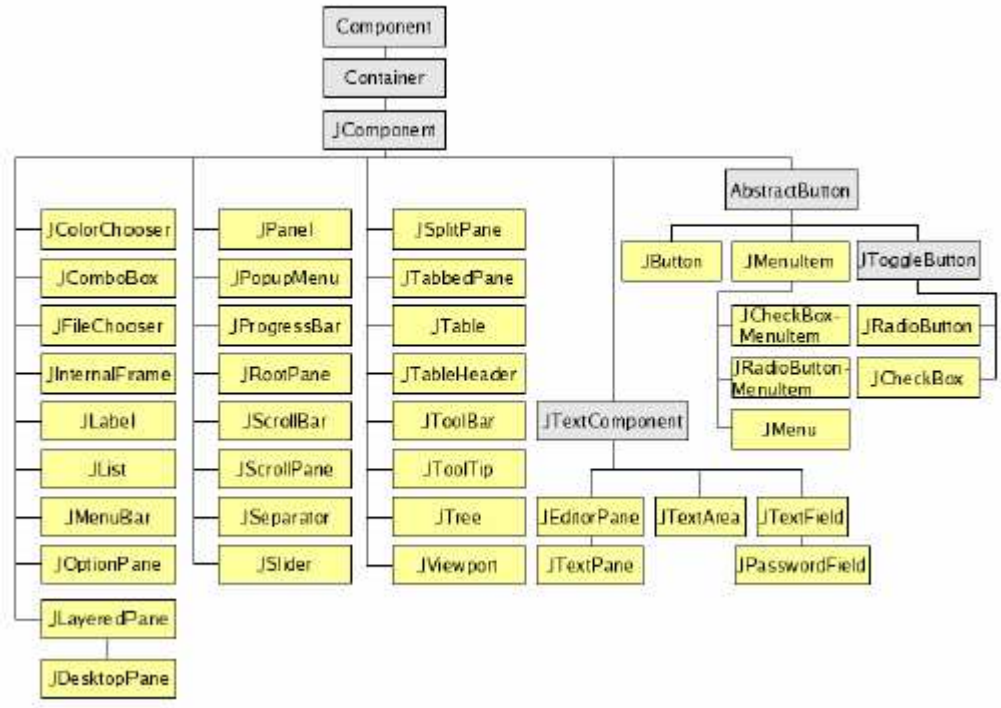

Figure 2.4: Swing component hierarchy

Swing disposes GUI resources and GUI components by the Java garbage collection mechanism.

2.4.2 SWT component hierarchy

The gray boxes represent are the most important classes for the SWT architecture. The based class is Widget which directly inherits from Object class. Control class is the super class for all the widgets while Item class is the super class for components or sub-components.

The Composite class is important since is the equivalent of AWT's container and is the super class of all components that allow children to be placed in them.

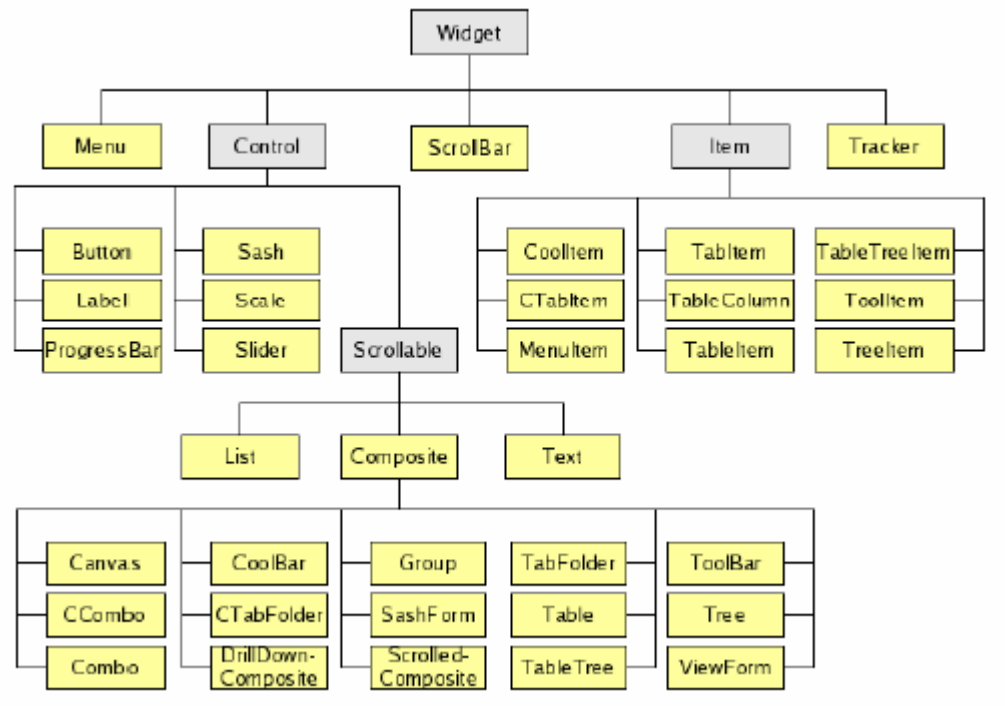

Figure 2.5: SWT component hierarchy

In contrast to Swing, SWT does not dispose GUI resources and GUI components by the Java garbage collection mechanism. System resources must be disposed of after using them in SWT manually. It is really important to dispose a SWT GUI component, such as Button or Text, when you are done using it. If the parent Composite is disposed all the children will be disposed too. Fonts and Colors are not part of a composite so it is necessary to dispose those objects manually.

2.4.3 Compare Swing versus SWT code

As we have seen so far, there two important differences between SWT and Swing. First the API has a different structure, and second, Swing auto-disposes components, where as SWT requires a manual dispose. In order to demonstrate these differences, a small piece of code is implemented giving an idea about how the implementation changes from Swing to SWT. Figure 2.6 shows how to create a window which contains a label (JLabel), button (JButton) and an action listener (ActionListener) for the button closing the application when is pressed.

First we created a simple java application consisting out of one class (HelloWorldSwing), which implements a main method, creating and displaying a frame (JFrame), with a label and button. Assuming familiarity with Java programming, we are only pointing out the Swing relevant parts of the code, which are marked in the code sample, corresponding to the numbers below:

A JFrame instance is created with "HelloWorldSwing" name.

A JLabel with “Hello World” as name is created, and added to the frame.

A JButton with “Click to close” as name is created and added to the frame. An ActionListener is implemented to close the application when the button is clicked (an action is performed on the button).

The *main* method executes the *run* method creating and showing the GUI.

It is to observe, that no explicit disposing code is required, but for thread safety the method creating and showing the GUI is invoked from the event-dispatching thread.

```
public class HelloWorldSwing {
    /**
     * Create the GUI and show it. For thread safety, this method
     * should be invoked from the event-dispatching thread.
     */

    private static void createAndShowGUI() {
        JFrame.setDefaultLookAndFeelDecorated(true);

1       JFrame frame = new JFrame("HelloWorldSwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

2       JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);

3       JButton button = new JButton("Click to close");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.exit(0);
            }
        });
        frame.setLayout(new FlowLayout());
        frame.add(label);
        frame.add(button);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        // Schedule a job for the event-dispatching thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
4           public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Figure 2.6: AWT/Swing sample code

Second, we created a simple java application consisting out of one class (HelloWorldSWT), which implements a main method, creating and displaying a frame (Shell), with a label and button. Assuming familiarity with Java programming, we are only pointing out the SWT relevant parts of the code, which are marked in the code sample, corresponding to the numbers below:

A Shell is instantiated which corresponds to a JFrame for Swing applications. The *shell* contains all the objects for an application.

A Label object with “HelloWorldSWT” as name is created and added to the shell.

A Button object with “Click to close” as name is created and added to the shell. A SelectionListener is implemented to close and dispose the application when the button is pressed.

As it was described in subchapter 2.4.2 the object must be explicitly disposed when the application is going to be closed.

```
public class HelloWorldSWT {
    /**
     * Create the GUI and show it. No thread safety needed.
     */

    public static void main(String[] args) {

        Display display = new Display();
1       final Shell shell = new Shell(display);
        shell.setText("HelloWorldSWT");
2       Label label = new Label(shell, SWT.NONE);
        label.setText("HelloWorldSWT");
3       Button close = new Button(shell, SWT.PUSH);
        close.setText("Click to close");
        close.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent e) {
                shell.dispose();
            }
        });
        shell.setDefaultButton(close);
        shell.setLayout(new RowLayout());
        shell.pack();
        shell.open();
4       while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
}
```

Figure 2.7: SWT sample code

Figure 2.8: Hello World example programs shows the result for both samples. On the left side the Swing application with the Java style, clearly not blending into the native appearance of the operating system. On the right side the SWT application with the native Windows look & feel style.

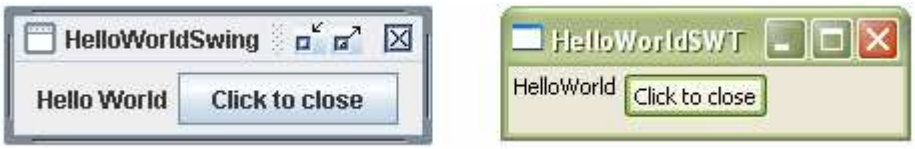

Figure 2.8: Hello World example programs

Anyway, it can be recognized in the examples, that the code is quite similar. But still there are some differences, e.g., about the way how *listeners* are managed for the different components, e.g., buttons, lists and so on. For further information on SWT programming refers to [7] where it is explained deeper the basic differences between AWT/Swing and SWT for a migration.

2.5 NetBeans versus Eclipse

NetBeans is developed by Sun Microsystems and offers different tools as the Eclipse Platform for plug-in development. Opposed to Eclipse, it is based on Swing instead of SWT.

Both tools provide similar functionality to the user. But for the plug-in developer NetBeans has some drawbacks. First it is based on Swing, which does not provide a native look & feel. This makes it inappropriate to reach out second goal. Furthermore, the lack of documentation for NetBeans makes it difficult to develop for it. It was not possible to decide, if all states goals, in particular, the third goal could be met. We could not find out in feasible time how to use the progress service, automatic update and help system can be utilized. The fifth goal is to improve performance, which we wanted to reach by utilizing SWT, was also not reachable. Additionally, NetBeans does not have as many third party plug-ins as Eclipse.

In the other hand, the Eclipse Platform is very well documented for plug-in developers and several books exist. Furthermore, SWT allows a improved GUI performance, and many third party plug-ins are available for Eclipse.

Because of these reasons, we decided that basing the VizzAnalyzer on Eclipse would be the best solution.

3 Requirements

We provide in this Section an overview about the new and changed requirements for the VizzAnalyzer. Since the majority of the requirements did not change for the new VizzAnalyzer 2.0, we limit ourselves to introducing the modifications. Refer to [1] for a complete description of the requirements.

First comes a brief outline of the users involved in the system, and then comes a short description of the additional features the system shall provide. We repeat the Use Cases capturing the functional requirements of the system, as already described in [1], and highlight the extensions.

3.1 Users

As described in [1], there are two users of the VizzAnalyzer, which are:

The end-user. He interacts with the VizzAnalyzer, performing activities like, loading graphs, and analyzing software. The End-User can interact with, control and change between perspectives and stop long running jobs. He can use the help system and update to newer program versions.

The developer. He extends the VizzAnalyzer Framework with new plug-ins and tools and maintains existing implementations, which are then provided over the update feature to the end-users.

3.2 Features

Adding to the existing features of the VizzAnalyzer, the following features shall be added or maintained in the new version of the VizzAnalyzer.

Feature 01	Capability to be extended with dynamic plug-ins
Description:	The developed software system has the capability of being extended with new and existing dynamic plug-ins, which integrates into the new SWT based GUI. It shall be possible to use already existing plug-ins having a Swing GUI, and new plug-ins having a SWT based GUI.
Feature 02	Capability to use features from Eclipse Platform
Description:	The developed software system has the capability of using new features from Eclipse Platform as progress bar, views and perspectives, product branding, updating. The system is capable to work with other Eclipse Plug-ins and tools, for instance, the help manager.
Feature 03	Capacity to maintain existing functionality from VizzAnalyzer
Description:	The developed software system has the capability to provide the same functionality as the existing VizzAnalyzer, including different actions as Load, Save, Cut, Copy Paste of graphs, Cut, Copy, Paste, and Scripting from the VizzAnalyzer Framework.

Table 1: Features

3.3 Use Cases

This section describes the existing Use Case Model and Use Cases, as described in [1], with the extensions resulting from the features described in the previous section.

3.3.1 Use Case Model

In the following we provide an overview about the different actors, showing in which way they are interacting with the system in various use cases. The Use Cases already handled by VizzAnalyzer version 1.0 are colored in gray. The white Use Cases have been added in the new version, which encompasses all depicted use cases.

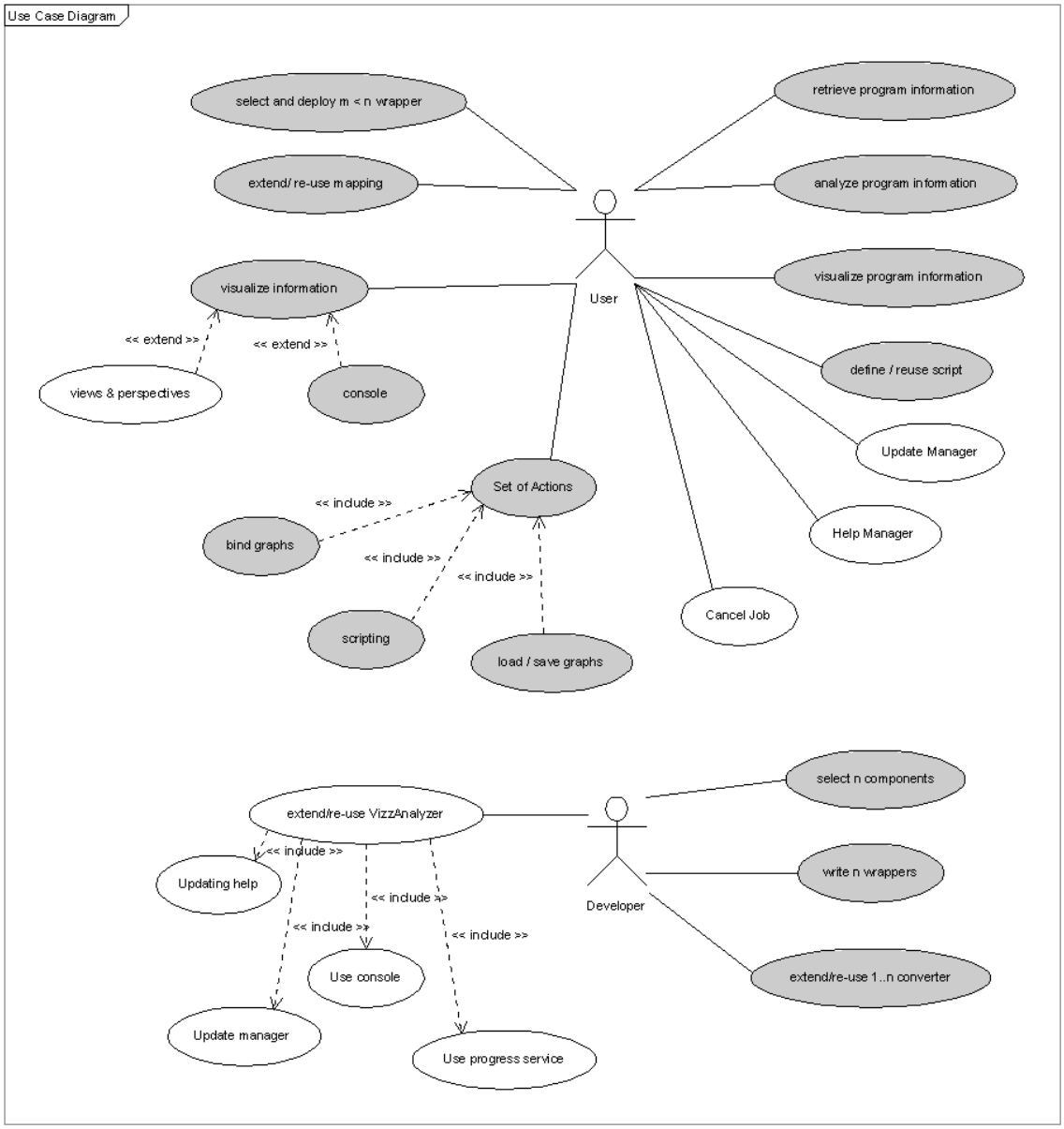

Figure 3.1: Use Case Model of VizzAnalyzer 2.0

The use cases colored in white describe the new use cases implemented. Gray use cases were already implemented for the old VizzAnalyzer and all of them have been re-used. Figure 3.1, shows the Use Cases Model for VizzAnalyzer 2.0. On its basis shall the requirements for the extension of the old VizzAnalyzer Framework be developed. To mention is, that the new VizzAnalyzer, as a part of the Eclipse Platform, can now be extended in two ways. First by using plug-ins and tools of the Eclipse Platform, and the second, extending the VizzAnalyzer Framework itself via plug-ins designed for it.

3.3.2 Use Cases

This subchapter describes the Uses Cases in detail. Each use case provides one or more scenarios that convey how the system should interact with the users to achieve a specific business goal or function. A use case has a title, goal, pre-condition, post-condition, trigger event that executes de use case, extensions and alternatives.

Use Case UC1	Set of actions Feature: F01, F03
Goal	Execution of actions already implemented in the VizzAnalyzer. Some actions are scripting and binding options from the VizzAnalyzer menu. Dynamic plug-ins are loaded on the menu bar.
Pre-condition	Product key activated
Post-condition	Any action from the menu from scripting and binding are executed.
Actors	User
Triggering event	User press on File, Edit, Script menu and choose any option from there. Different actions are available. The user can analyze, visualize, read information from dynamic plug-ins
Description	The user needs to select one option from the menus. Some actions need a graph already loaded, others as the scripting menu is not necessary. From file menu is possible to load graphs and save them. From Edit menu is possible Cut, Copy and Paste. From Script menu is possible to use the scripting.
Extensions	-
Alternatives	User does not have a correct key and the application can not run.

Use Case UC2	Views & Perspectives Feature: F02, F03
Goal	View the information from a graph loaded in the system, including its nodes, edges and all properties.
Pre-condition	Graph loaded.
Post-condition	Graph properties, node and edge information is shown by a dedicated view and perspective.
Actors	User
Triggering event	User selects view or perspectives with a graph selected in menu or toolbar.
Description	The user selects a graph. The system loads the graph. The system shows the information is in different tables. The user selects between different perspectives displaying the available information.
Extensions	-
Alternatives	-

Use Case UC3	Cancel Job	Feature: F02
Goal	A job throw by the system is cancelled by the user.	
Pre-condition	A long running job is running.	
Post-condition	The job is cancelled.	
Actors	User	
Triggering event	The system executes a background job.	
Description	The system has executed a long running job and the user can cancel it if it is necessary. Otherwise, the job is executed until it finishes.	
Extensions	-	
Alternatives	-	

Use Case UC4	Update system	Feature: F02
Goal	Update the system if a new version is available on the web.	
Pre-condition	The computer is connected to the internet, a newer version is available	
Post-condition	The software is updated to the newest version, the application restarts	
Actors	User	
Triggering event	The user press on Help menu, Update option.	
Description	The user starts the update feature. The system connects to the web and checks the version from the user with the final released version on the update page. If a new version of the software is available, the system downloads and installs the new version.	
Extensions	-	
Alternatives	3a. If no newer version is available, the update function terminates.	

Use Case UC5	Help system	Feature: F02
Goal	The user is provided with help and additional information about the system or features of the system.	
Pre-condition	The help system is available.	
Post-condition	The resources occupied by the help system are freed.	
Actors	User	
Triggering event	The user activates the help system using the help menu.	
Description	The user opens the help system over the help menu. The user browses the help system over the index, search or browse functionality. The system displays the associated help information. The user ends the help system, or continues with 2.	
Extensions	-	
Alternatives	3a. No help is associated with a specific query. No information found message is displayed.	

Use Case UC6	Use Console	Feature: F02
Goal	The developer has access to a console view where it is possible to give feedback to the end-user, about running operations.	
Pre-condition	The developer needs to have the source code installed.	
Post-condition	The developers can show messages about the VizzAnalyzer in the console.	
Actors	Developer	
Triggering event	A new component or plug-in is added to the VizzAnalyzer framework.	
Description	The developer opens any class from the VizzAnalyzer application. The developer calls the method to show a message up. The system displays the message on that point when the application is running. The developer/user receives the feedback from the operation.	
Extensions	-	
Alternatives	-	

Use Case UC7	Use Progress Service	Feature: F02
Goal	The developer provides the end-user with progress information about longer running tasks.	
Pre-condition	The developer needs to have the source code installed.	
Post-condition	The developer implements a progress service on the VizzAnalyzer for any operation.	
Actors	Developer	
Triggering event	A new component or plug-in is added to the VizzAnalyzer framework.	
Description	The developer uses the API providing a progress service for an operation. The system displays a progress bar when the operation (implementing a progress service) is running. The developer/user receive see the progress and they wait until the operation is over.	
Extensions	-	
Alternatives	3a The developer/user cancel the operation in progress.	

Use Case UC8	Updating the Update Manager	Feature: F02
Goal	Publish updated version of the VizzAnalyzer to end-users.	
Pre-condition	The developer needs to have the source code installed.	
Post-condition	The update site hosting the VizzAnalyzer is updated with a new build. The end-users can download it.	
Actors	Developer	
Triggering event	A new component or plug-in is added to the VizzAnalyzer framework, or the current code base has been modified.	
Description	The developer builds a new version of the VizzAnalyzer. The developer publishes the new build on the download site.	
Extensions	-	
Alternatives	-	

Use Case UC9	Updating the Help Manager Feature: F02
Goal	Update the help content.
Pre-condition	The developer needs to have the source code installed.
Post-condition	The help content giving more information about some topic of the VizzAnalyzer is updated.
Actors	Developer
Triggering event	A new component or plug-in is added to the VizzAnalyzer framework, or the current code base has been modified.
Description	The developer updates the content of the help system. The developer builds the new help system. The developer publishes the new help content with the next build.
Extensions	-
Alternatives	3a The user/developer does not use the help manager.

Use Case UC10	Extend/re-use VizzAnalyzer Feature: F02
Goal	Extent the VizzAnalyzer framework with a new plug-in.
Pre-condition	The developer needs to have the source code installed.
Post-condition	The developers developed a new plug-in for the VizzAnalyzer.
Actors	Developer
Triggering event	A new component or plug-in is needed.
Description	The developer implements the new functionality. The developer extends the VizzAnalyzer with this plug-in. The developer includes these features in the next build or as separate features for download.
Extensions	-
Alternatives	-

Table 2: Use Cases

3.4 Functional requirements

This chapter describes the important new functional requirements which add new behavior to the VizzAnalyzer. This behavior is expressed as services, tasks or functions which the system is required to perform, and associated with the corresponding Use Cases.

Requirement R01	Help manager Event / Use Case: UC5
Description:	The system shall provide a help system. The user can browse and search the help topics. The help system is extensible with new help topics.
Rationale:	The VizzAnalyzer is becoming a complex application. It is necessary to aid the end-user with help and additional information about the functionality of the system, to improve its usefulness and acceptance in the user community. Since the VizzAnalyzer is extended with new plug-ins, it is necessary to extend the help system with information about these new

	features.
Fit Criterion:	The user is able to have access to the help of the system. Browsing and querying of the help topics is possible using the search engine of the help system plug-in. The content of the help topics can be extended.
Requirement R02	Update manager Event / Use Case: UC4
Description:	The system shall provide an update manager. The user shall be able to start an update process. The system shall detect if the current version is the same than the latest released published on the update side. In case user has an old version the system inform the user letting him to download the latest released.
Rationale:	The VizzAnalyzer framework is steadily growing with new features and plug-ins. It is necessary to provide an easy way to publish updates, bug fixes, and new plug-ins to the end-user, without reinstalling each time and update is available.
Fit Criterion:	The user is able to update the software system if a new version or extensions are published on the update web site.
Requirement R03	Views and perspectives Event / Use Case: UC2
Description:	The system shall provide different views and perspectives allowing the user to view and interact with the information loaded as graphs.
Rationale:	To be able to work with analysis results (front-end or other analysis), it is necessary to view this information to the end-user, to allow him making decisions, and to understand and comprehend the information.
Fit Criterion:	The user is able to see the graphs resulting from front-ends, or loading of graphs, including all associated information like graph, node and edge properties.
Requirement R04	Console Event / Use Case: UC2
Description:	The system shall provide a console output providing feedback to the user.
Rationale:	Different parts of the VizzAnalyzer Framework and plug-ins use no graphic output over dialogs or windows, but console output, for informing the user about progress or problems. In the previous version the VizzAnalyzer used the System console or terminal window for output. This is not available in Eclipse, so an alternative needs to be provided.
Fit Criterion:	The user is able to see the textual messages printed by different parts of the application or plug-ins in a dedicated console window.
Requirement R05	Progress Service Event / Use Case: UC3
Description:	The system shall provide a progress service. The progress service can execute a job in the fore- or background for longer-running jobs. The user sees the progress of the current job and for longer-running jobs is allowed to be canceled by the user.

Rationale:	Certain analysis or other actions performed by the end-user can take long time. The user needs to be informed about the progress and must be able to cancel actions.
Fit Criterion:	The system provides progress information to the user, if a certain task takes more than 2 seconds. The user can cancel the task over a Cancel button provided by the progress dialog. The User can alternatively change the task into a background task.
Requirement R06	Actions Event / Use Case: UC1
Description:	Already existing functionality of the VizzAnalyzer shall be maintained.
Rationale:	The system is able to execute different actions as Copy, Paste, Cut, Load and save graphs. This functionality is an important part of the VizzAnalyzer, and needs to be available also in the new version, since it is necessary for working in the desired way.
Fit Criterion:	The end-user has the same actions available as in the previous VizzAnalyzer version.
Requirement R07	Check key product Event / Use Case: UC1
Description:	The system shall require a license key on first start-up. If the product is not registered the user shall provide a new key code to activate the software. This feature is already implemented in the VizzAnalyzer and shall be provided.
Rationale:	The authors of the VizzAnalyzer put a lot of effort into the VizzAnalyzer. They want to control its use, and prevent commercial use of it, since it is only free for non-commercial use.
Fit Criterion:	The user needs to introduce a key product the first time. The system validates the key and allows using the software.
Requirement R08	Reading dynamic plug-ins Event / Use Case: UC1
Description:	The system shall provide menus for the different dynamic plug-ins loaded by the VizzAnalyzer Framework.
Rationale:	The main purpose of the VizzAnalyzer Framework is to integrate different plug-ins (analysis, front-end, visualization) over a common data structure. They are made available over the menu structure of the VizzAnalyzer GUI. This feature must be maintained in the new version.
Fit Criterion:	The user is able to access over a menu the different plug-ins.

Table 3: Requirements

4 Architecture

We described the requirements on the new VizzAnalyzer in the previous section. Now we shall explain how they are transformed into an architecture for the new system. For this purpose we discuss the general architecture of an Eclipse application and compare it with the original VizzAnalyzer architecture. By comparing both architectures we will elicit what changes we need to make to the current VizzAnalyzer architecture, and how the Eclipse architecture is utilized.

Therefore, we focus first on a description of the Eclipse architecture and thereby explaining what a plug-in in terms of Eclipse is, and how it relates to the Eclipse philosophy.

Second we describe the new VizzAnalyzer architecture, showing the interface between Eclipse and the VizzAnalyzer, highlighting the new or changed parts connecting the existing functionality with the new GUI.

4.1 The Eclipse architecture

The complete Eclipse Project is depicted in Figure 4.1: Eclipse architecture. The ellipses show what parts of the architecture have been used in the context of this thesis. These are namely the User Interface consisting out of the Workbench, JFace and SWT, PDE and the Core Runtime. The basic unit of function in this framework is called a *plug-in* – the unit of modularity in Eclipse.

To be complete in our description of the architecture, we are also explaining briefly the parts not utilized by us. These are the *Java Development Tools* (JDT) which allows users to write compile, test, debug, and edit programs written in the Java programming language. It is a set of plug-ins that adds Java specific behavior to the generic platform resource model and contributes Java specific views, editors and actions to the workbench. The *Workspace* is the plug-in responsible of managing the user's resources. Projects that user create, files included on them and other resources.

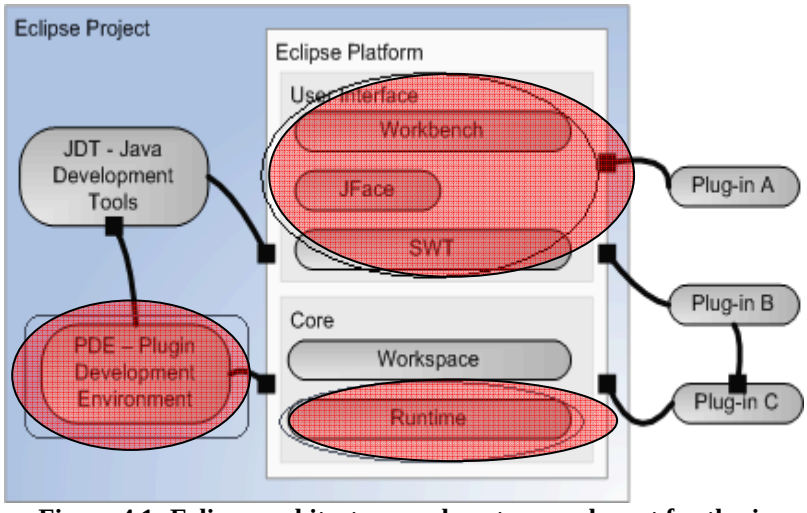

Figure 4.1: Eclipse architecture, red parts are relevant for thesis

The parts concerning us, since they are important for the evolution of the VizzAnalyzer architecture, are the PDE (Plug-in Development Environment), Workbench, JFace, SWT and the Runtime, as highlighted in Figure 4.1: Eclipse architecture. They are explained as follows.

The *Workbench* provides a Graphical User Interface (GUI) to Eclipse using SWT and JFace. The *Platform Runtime* is the kernel that discovers what plug-ins are installed, creating a registry of information about them. The *Plug-in Development Environment* (PDE) is mainly interesting for developers who want to extend Eclipse that is us. The PDE includes tools for exporting based on a wizard or ANT tools for big applications that needs a control versioning for instance.

This architecture is so flexible, that any developer can assemble a collection of plug-ins from the Eclipse base, while third parties can assemble their own plug-ins into Eclipse base meanwhile. All created plug-ins can interact with each other since they are using the same standardized interfaces. How this works is described in the following section.

4.1.1 The Eclipse Plug-in Model

The Eclipse runtime provides an infrastructure to support the activation and operation of a set of plug-ins allowing them all to work together. All Eclipse installations include a *plugins* folder where individual plug-ins are deployed. A plug-in is described as a XML based manifest file, called *plugin.xml*. The manifest file tells the Eclipse runtime what it needs to know to activate that plug-in.

Here is the minimal plug-in manifest file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

  <extension
      id="application"
      point="org.eclipse.core.runtime.applications">
    <application>
      <run
          class="sampleplugin.Application">
      </run>
    </application>
  </extension>

</plugin>
```

Figure 4.2: Plugin.xml

The *extension point* informs there is a dependency with *org.eclipse.core.runtime.applications* package and the class who runs the application is location on package *sampleplugin* and the main class is *Application*. This mechanism is going to be used for integrating the VizzAnalyzer with Eclipse.

4.1.2 Extension

The process of adding some processing element or elements is known as *extension*. This process is not restricted for UI elements. Any plug-in may allow other plug-ins to extend it by adding processing elements. A plug-in may allow itself be augmented by different kind of extensions. *Extension* and *extension-point* are standard Eclipse plug-in terminology.

It is important to know what dependencies there are for any extension or between plug-ins, otherwise a runtime error occurs when the application starts. Eclipse 3.0

provides a view where is possible to see the dependencies between plug-ins. For further information about Eclipse architecture see [9] [10].

4.2 The VizzAnalyzer architecture

After providing an overview about the Eclipse plug-in architecture and extension mechanism, we shall have a closer look onto the VizzAnalyzer version 1.0 architecture as it is depicted on Figure 4.3: The VizzAnalyzer Framework's architecture.

The Reverse-Engineering Framework is responsible for communicating information between the different reverse engineering components connected to the framework. To these components belong so called hot and frozen spots. The frozen-spots are part of the framework, and not supposed to be exchanged frequently. The hot-spots are plug-ins which can be dynamically loaded into the framework. These include converters, transforming foreign to native data formats, mapping-files, mapping information stored in the internal data structure, and wrappers connecting retrieval, analysis and visualization plug-ins to the framework. These shall not be touched by our efforts, and are therefore not of interest.

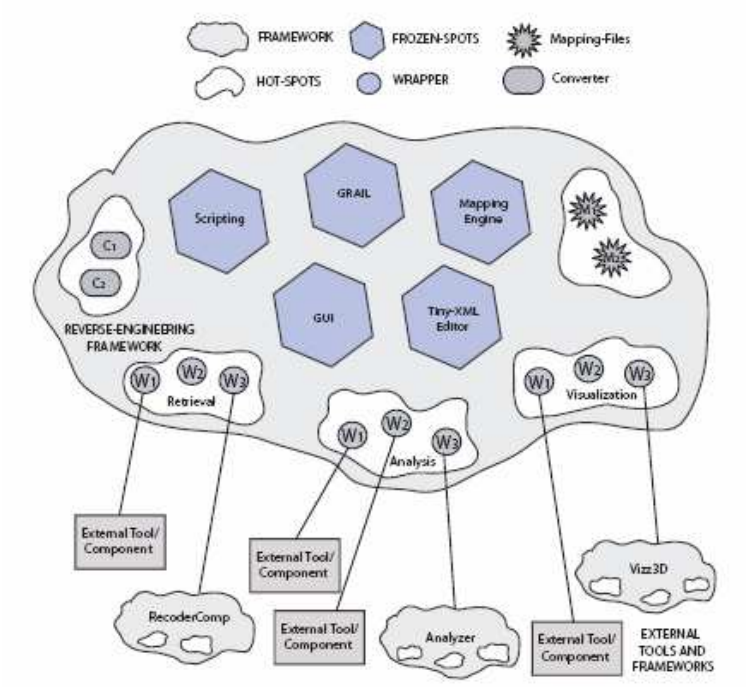

Figure 4.3: The VizzAnalyzer Framework's architecture

The more interesting parts are the so called frozen-spots which are described in the following in some more detail:

GRAIL is a graph library and it is used as an internal data representation for the VizzAnalyzer Framework. The graph has information attached as nodes and edges and the graph itself.

Mapping Engine. The framework architecture distinguishes the domains of software analysis and information visualization. The mapping engine allows mapping the models of the analysis to the visualization domain.

Tiny-XML Editor. The editor, it is a stand-alone tool for configuring XML documents. Used to configure mapping files and other configuration files online.

Scripting. In order to perform a sequence of operations between plugged components repeatedly, it is used Java BeanShell, a free Java source interpreter with object scripting language features.

GUI. To allow a fast and easy use of the VizzAnalyzer Framework it comes with a predefined GUI. This is fulfilled with the contents of the hot-spots of the framework. Depending on the components plugged to the framework, the main menu differs providing specific items [1].

The part mainly affected by the refactoring of the VizzAnalyzer version 1.0 is its GUI frozen-spot. It shall be replaced by the Eclipse GUI, and the VizzAnalyzer itself, shall become a plug-in into the Eclipse Platform. The parts of the VizzAnalyzer which are not related to the interfaces to Eclipse remain unchanged and function as before. The VizzAnalyzer has some plug-ins which have their own GUI (Swing), which is not controlled by the VizzAnalyzer. Their GUI, which are mainly internal dialogs written in Swing are not being the refactored by this thesis. They will be shown as Swing, until the plug-in authors adapt them. The architecture developed for the VizzAnalyzer version 2.0 is depicted in Figure 4.4.

That means, the VizzAnalyzer 2.0 keeps its internal architecture, but it is now coupled with the Eclipse Platform architecture. The GUI is removed from the old architecture being replaced by a plug-in for the Eclipse Platform.

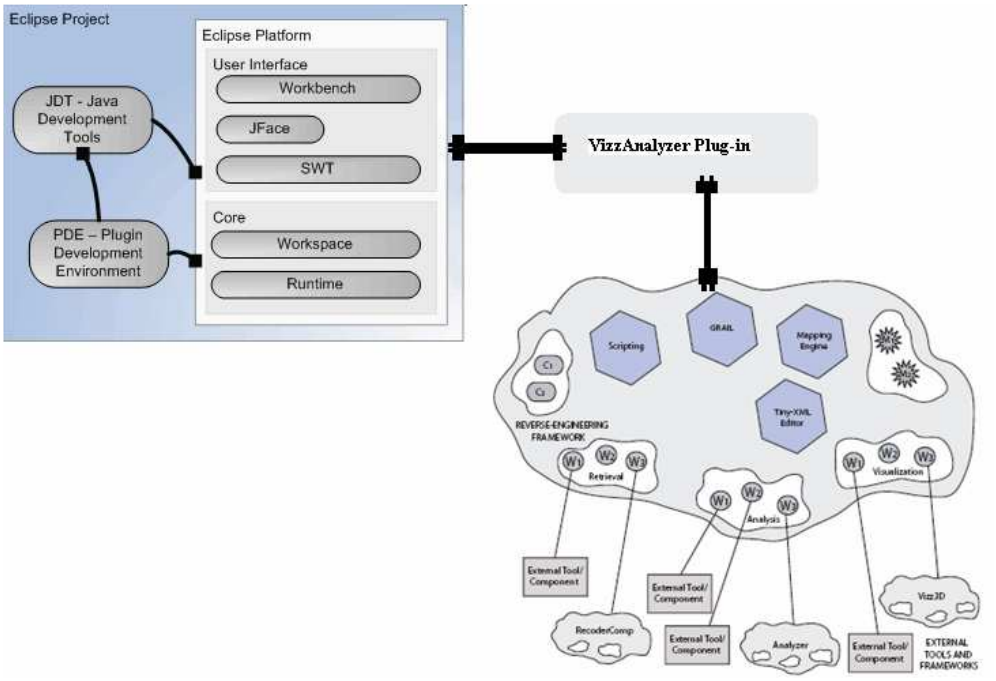

Figure 4.4: The VizzAnalyzer 2.0 architecture

5 Design and implementation

After explaining the architecture for combing the VizzAnalyzer with the Eclipse platform in the previous chapter, we will explain in the following sections how the original design and implementation of the VizzAnalyzer has been modified to satisfy the requirements of the new architecture. While Section 5.1 describes the reengineering of the VizzAnalyzer startup procedure, the rest of the design and implementation of VizzAnalyzer 1.0 is not extensively described, but referenced from Section 5.2, when appropriate, to compare with the new design and implementation of VizzAnalyzer 2.0.

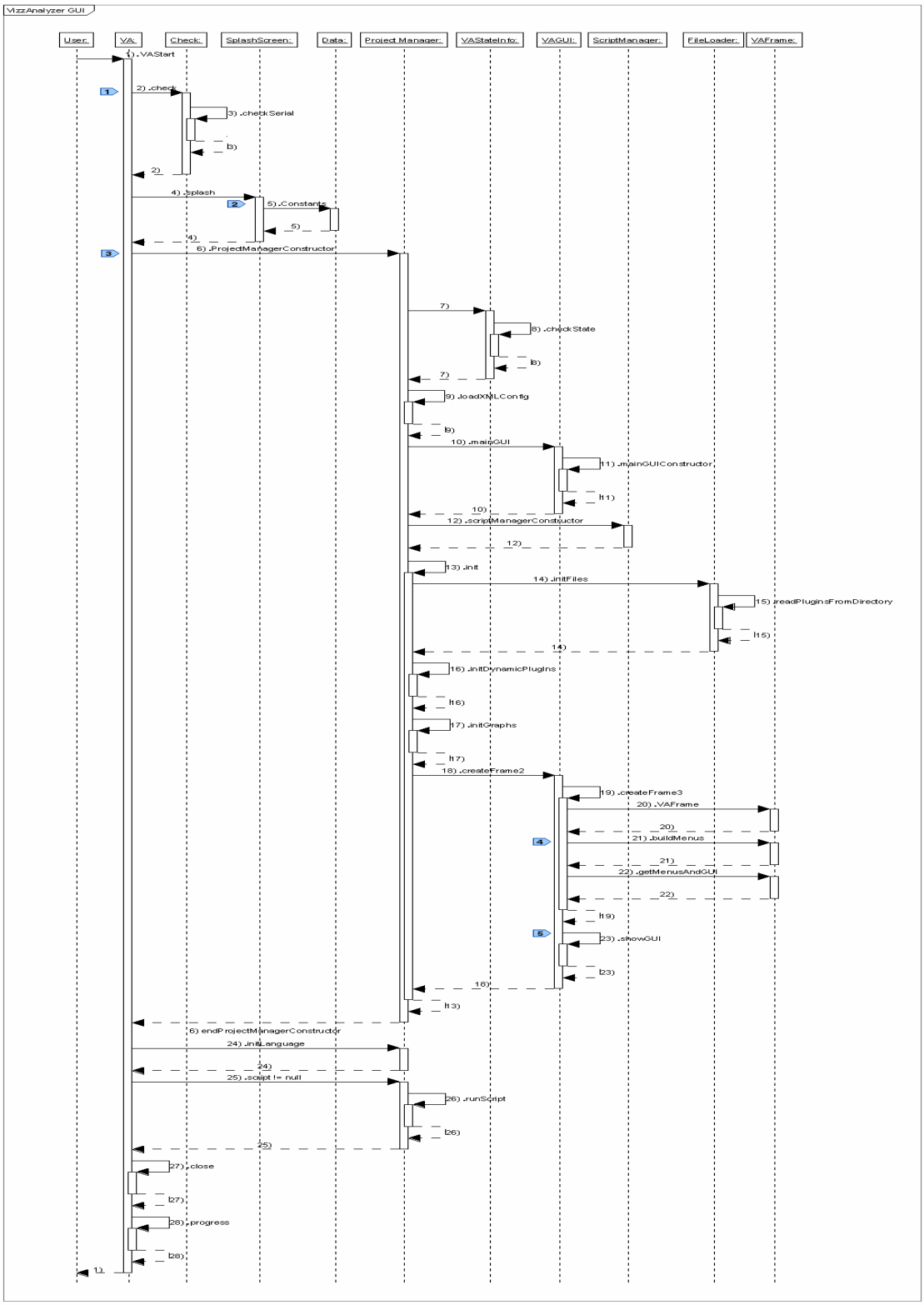

Figure 5.1: GUI design from VizzAnalyzer 1.0

5.1 The VizzAnalyzer 1.0

The VizzAnalyzer 1.0 was based on Swing. It was a stand-alone application having core.VA as its main class containing the *main()* method as entry point. Starting the application the following happens. See sequence diagram of the startup procedure in Figure 5.1.

1 Check of the license key.
2 Loading of constants and configurations needed for the GUI generation
3 A *ProjectManager* object is created, which is responsible to build the main GUI.
4 It also loads the dynamic plug-ins adding them to the *VAFrame*.
5 GUI for the VizzAnalyzer is shown

This process needed to be well understood, since it needed to be transferred to the Eclipse plug-in start-up procedure. Other parts of the VizzAnalyzer and Design are for now not relevant, and will be discussed when appropriate.

5.2 The VizzAnalyzer 2.0

The new version of VizzAnalyzer (version 2.0), is not a stand-alone application, but a plug-in to a framework. Therefore the entry point is not any longer the main method of the core.VA class, but the application developed must implement the *IPlatformRunnable* interface, which it is the equivalent to the *main()* method in Java.

The plug-in had now to implement the tasks previously executed by the main method, e.g., perform the license checking and loading the plug-ins. The main part of the GUI is already provided by the Eclipse framework. Therefore it is "only" necessary to add the VizzAnalyzer specific elements, like some menus and plug-in entries.

The VizzAnalyzer 2.0 follows a Plug-in Model as described in 4.1.1, therefore a *plugin.xml* is defined as shown in Figure 5.2. The name of the Eclipse plug-in is specified as id, as it marked by symbol 1. The main class executed as entry point, when Eclipse starts the framework, is marked by the symbol 2. Note it is the full qualified class name including package names. The class has to be in the plug-in folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
  1     id="VizzAnalyzerApplication"
      point="org.eclipse.core.runtime.applications">
    <application>
      <run
  2   class="se.vxu.msi.vizzanalyzer.VizzAnalyzerApplication">
      </run>
    </application>
  </extension>
</plugin>
```

Figure 5.2: VizzAnalyzer plugin.xml

This means in particular, that the *VizzAnalyzerApplication*, which implements the *IPlatformRunnable* interface, is loaded and executed by the Eclipse Platform, thereby creating a Display and starting an Eclipse Workbench. During this process the *VizzAnalyzerWorkbenchAdvisor* tells the Workbench how to behave. The *VizzAnalyzerWorkbenchAdvisor* identifies the initial perspective to be shown. The *VizzAnalyzerPerspective* implements the *PerspectiveFactory* interface where a

createInitialLayout method must be implemented defining the perspective layout. Only one perspective can be identified as initial perspective.

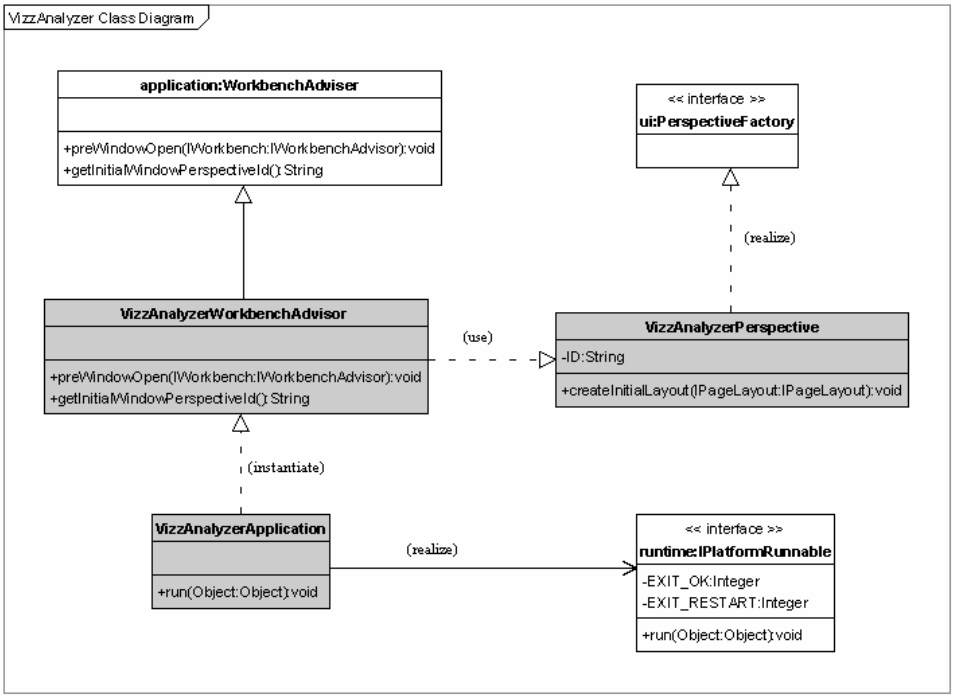

Figure 5.3: VizzAnalyzer Class diagram

The white classes shown in Figure 5.3 are provided by the Eclipse Platform and the gray classes are implemented for the current VizzAnalyzer 2.0 being the base architecture of the plug-in and the core of the VizzAnalyzer Plug-in discussed in Section 4.2. They are necessary for the correct implementation under SWT and Eclipse.

The *VizzAnalyzerApplication* class substitutes the core.VA class from the VizzAnalyzer 1.0 implementation, performing the tasks discussed in the Section 5.1 (sequence diagram. The other two classes support it. Partially could the code from the main() method of the core.VA class be reused, partially it needed to be adapted.

The *VizzAnalyzerActionBarAdvisor* is responsible to create the actions needed for the window and position them. They are instantiated using *creationActionBarAdvisor* method on *VizzAnalyzerWorkbenchWindowAdvisor*. It is also responsible for instantiating *VizzAnalyzerActionFactory* being responsible for centralizing and managing all the different types of actions of the application.

Since Eclipse uses other implementations of the Action classes as Java Swing, adaptations where necessary, for integrating the menu actions into the SWT based user interface. Therefore we have to distinguish two different types of actions:

Actions from VizzAnalyzer 1.0 are based on Swing and they implement the *javax.swing.Action* interface.

Actions from VizzAnalyzer 2.0 are based on SWT and the implement the *org.eclipse.jface.action* interface.

The implementation corresponding to SWT can be seen in Figure 5.4.

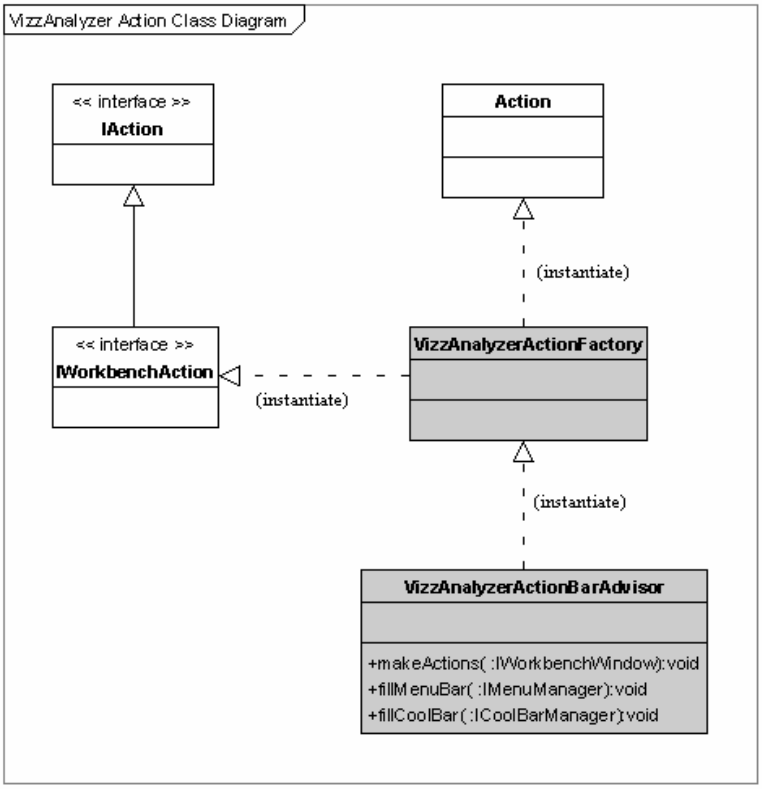

Figure 5.4: Action Class Diagram

The new start-up sequence, implemented by the VizzAnalyzerApplication class, is shown in Figure 5.5. As for VizzAnalyzer 1.0, the license key is checked. Note, that the user interface for checking the key product is still using Swing, since it is not part of the context of this thesis. Furthermore, the menu actions are installed and the dynamic plug-ins are loaded. After the startup-sequence is finished, the application is waiting for user interaction. In detail the following takes place:

When the user executes the VizzAnalyzer 2.0, the first class executed is VizzAnalyzerApplication (numbers correspond to numbers in diagram, Figure 5.5):

The *run* method is called starting the flow and checking the key of the product.

The application executes a *new* VizzAnalyzerWorkbenchWindow instance. Windows advisors are consulted at various points in the lifecycle of a window.

At this point the *preWindowOpen* method from VizzAnalyzerWorkbenchWindow calls VizzAnalyzerWorkbenchWindowAdvisor.

In the *preWindowOpen* method, it sets the initials customizations as size, title window, cool bars, status and other configuration window options.

The *getInitialPerspectiveId* method loads the initial perspective for the system. This is new, needed to show the initial perspective when the application already started. Different perspectives could be available but only one can show at the beginning up.

VizzAnalyzerWorkbenchWindow executes the *createActionBarAdvisor* method on VizzAnalyzerWorkbenchWindowAdvisor.

A *new* instance of *VizzAnalyzerActionBarAdvisor* is created. This is new, needed to build all the actions on SWT.

VizzAnalyzerWorkbenchWindow execute the *makeActions* method.

The *VizzAnalyzerActionBarAdvisor* load all the actions on the system. Fill the menu bar of the VizzAnalyzer, has been adapted to deal with the SWT menu system

and build the dynamic plug-ins from the VizzAnalzyer 1.0. Fill the cool bar of the VizzAnalyzer and fill the status line of the VizzAnalyzer are optional menus. The most common options are built there giving a better end-user experience.

The VizzAnalyzerWorkbenchWindow calls the *fillMenuBar* method.

The VizzAnalyzerWorkbenchWindow calls the *fillCoolBar* method.

The VizzAnalyzerWorkbenchWindow calls the *fillStatusLine* method.

The *postWindowOpen* method performances other actions when the window has been opened. The application is waiting for any interaction between the user and the system, all listeners and actions are loaded.

If the user closes the application, the *dispose* method from VizzAnalyzerWorkbenchWindow is called removing all the objects depending on it.

Following the flow the VizzAnalyzerWorkbenchWindow calls the *dispose* method and the VizzAnalyzerWorkbenchWindowAdvisor is removed.

The startup is completely new compared with the VizzAnalyzer 1.0, yet it performs the same VizzAnalyzer related tasks. The process follows the plug-in model from Eclipse. The *ProjectManager* object has been be re-used because it is an important part for the VizzAnalyzer 1.0 and the startup for the dynamic plug-ins are implemented there. *ProjectManager* object could keep much of the original code which has a high dependency to the GUI, scripting, and dynamic plug-ins. Where possible, code has been re-used and adapted if necessary. This was eased, since everything is managed with events (event based architecture).

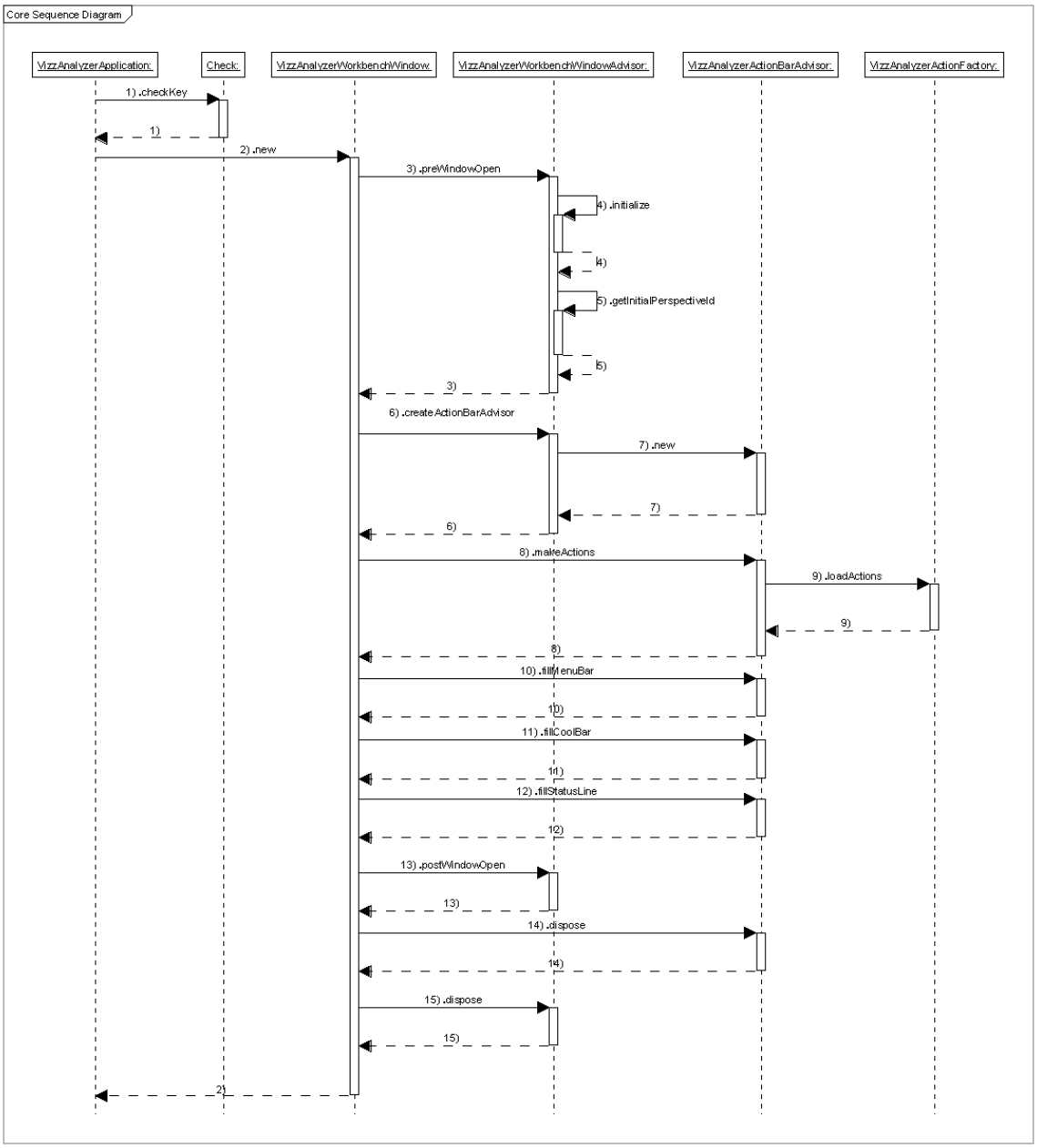

Figure 5.5: VizzAnalyzer Core Sequence Diagram

5.3 VizzAnalyzer extensions

We introduced in section 4.1.2 term *extension* and we explained that a plug-in can have dependencies with other plug-ins or extend from them. VizzAnalyzer 2.0 as an Eclipse plug-in is built as a *product* and *application*, using a wizard which is configured through the plugin.xml. Figure 5.6 shows the wizard view on the extensions of the plug-in. These extensions realize the views, perspectives and help features.

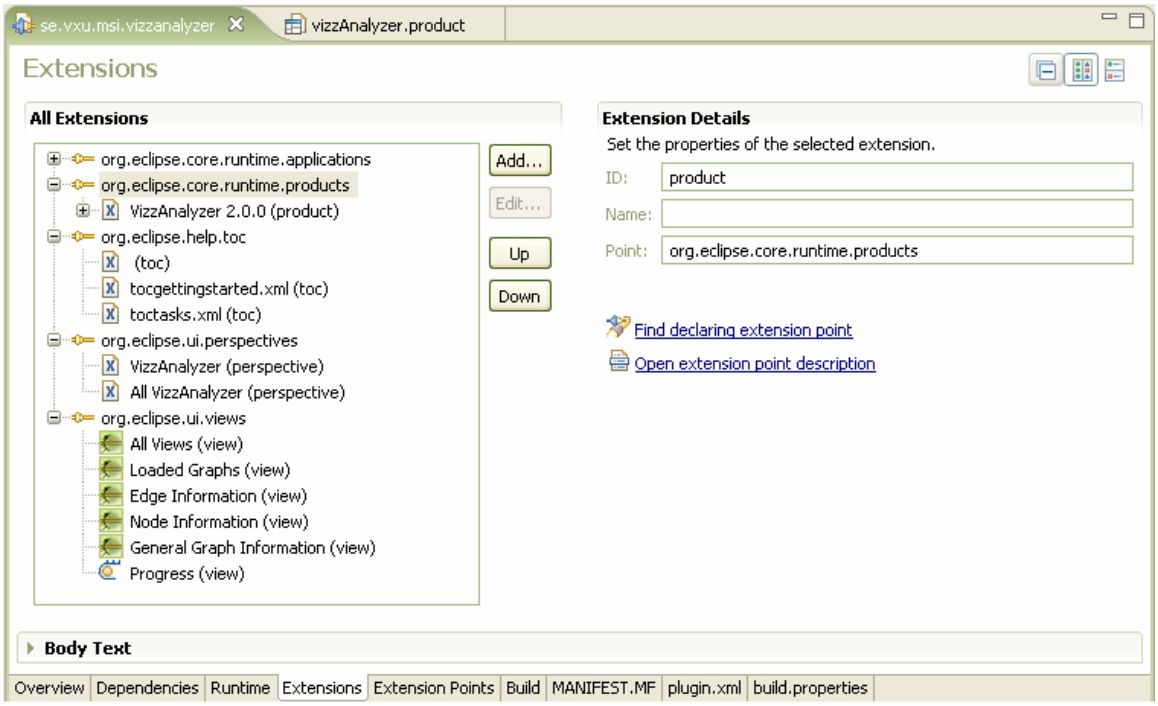

Figure 5.6: VizzAnalyzer Extensions

The Extensions tab gives and overview about all the extensions used by the VizzAnalyzer. E.g., the perspectives (org.eclipse.ui.perspectives) or views (org.eclipse.ui.views) are part of the User Interface.

If a plug-in extends from a *product* a list of plug-ins or fragments is needed. The extensions are referred over the libraries your plug-in uses, and they must be included (distributed) in your plug-in, otherwise a runtime error occurs when the application runs.

Figure 5.7 shows some of the plug-ins and dependencies necessary for VizzAnalyzer 2.0. Visible in this view is the required SWT library which is necessary to implement the UI under the Workbench of Eclipse. Others plug-ins are added for the Help and Update manager, more about them in later chapters.

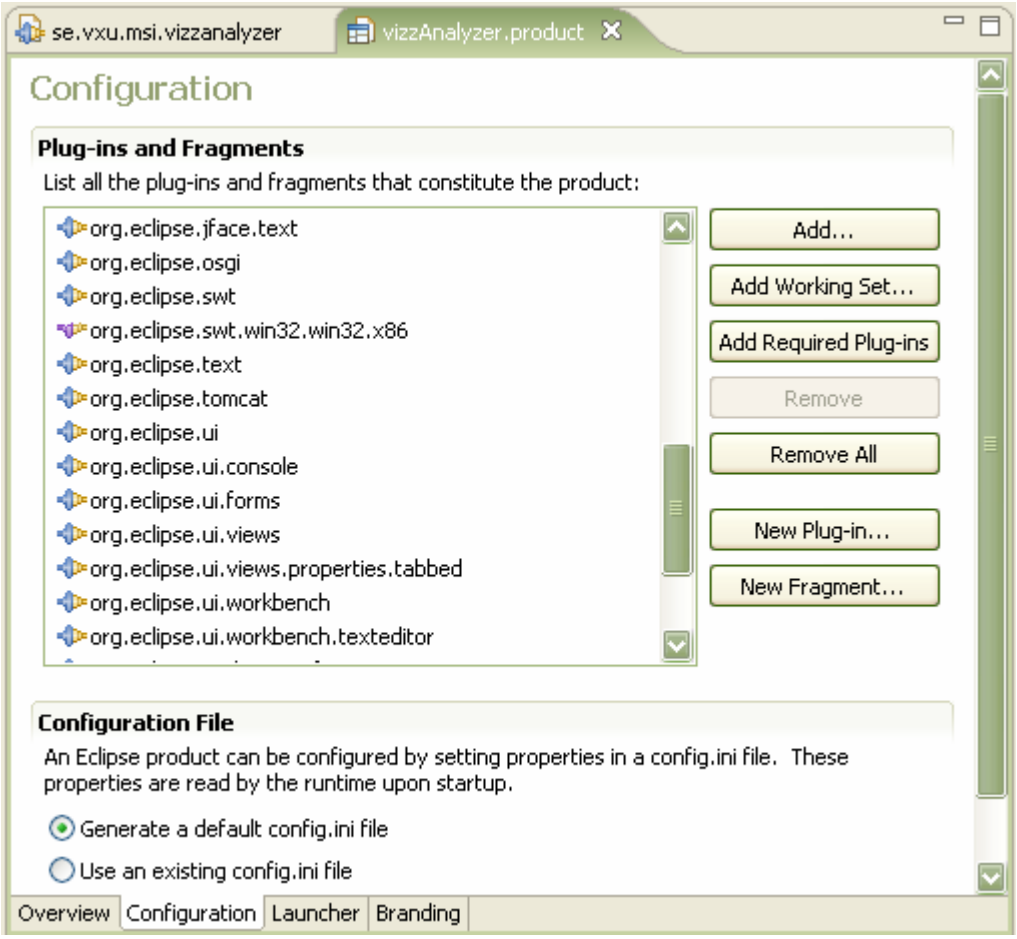

Figure 5.7: VizzAnalyzer plug-ins and fragments

Because it is really difficult to know which dependencies a plug-in has, from Eclipse 3.0 on a dependency analysis utility is included that shows the plug-in dependency hierarchy helping the developer to save time. Figure 5.8 shows the dependencies for the selected plug-in or component marked with a circle.

For instance, selecting *org.eclipse.core.runtime* displays a list of required plug-ins, showing that it is directly depending from *org.eclipse.core.contentype*, *org.eclipse.core.jobs* and so on. The same holds for *org.eclipse.ui* and *org.eclipse.ui.console* for instance.

The Plug-in dependencies wizard must be used to ensure that the plug-ins needed by the core of any plug-in, are provided. We did not use this tool during the first part of the implementation which resulted in many problems because the plug-in dependency hierarchy was not correct and the VizzAnalyzer could not startup.

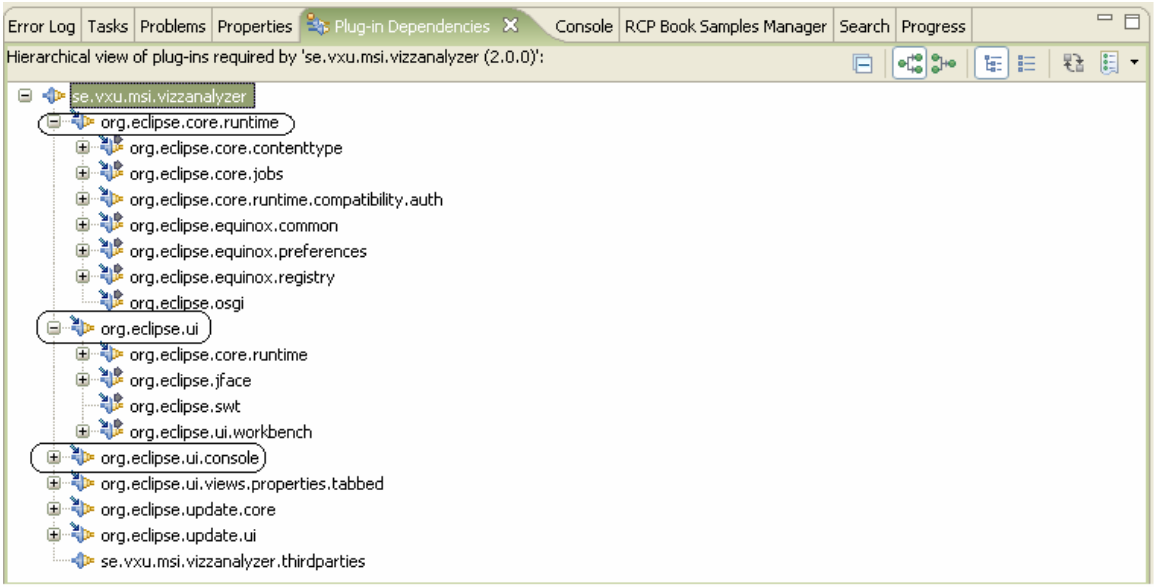

Figure 5.8: Plug-in dependencies

5.4 VizzAnalyzer tree viewer

One of the central parts of the old and new VizzAnalyzer GUI is a tree view displaying the currently loaded data and view graphs for managing them. While in the old GUI it was implemented using a JTreeView Swing component, this essential part had to be reimplementation for the new version using SWT. Figure 5.9: Graph Model Class Diagram shows the basic model designed for storing graphs in the Graph View tree viewer. The gray classes are provided by JFace while the white classes are implemented for the new VizzAnalyzer architecture.

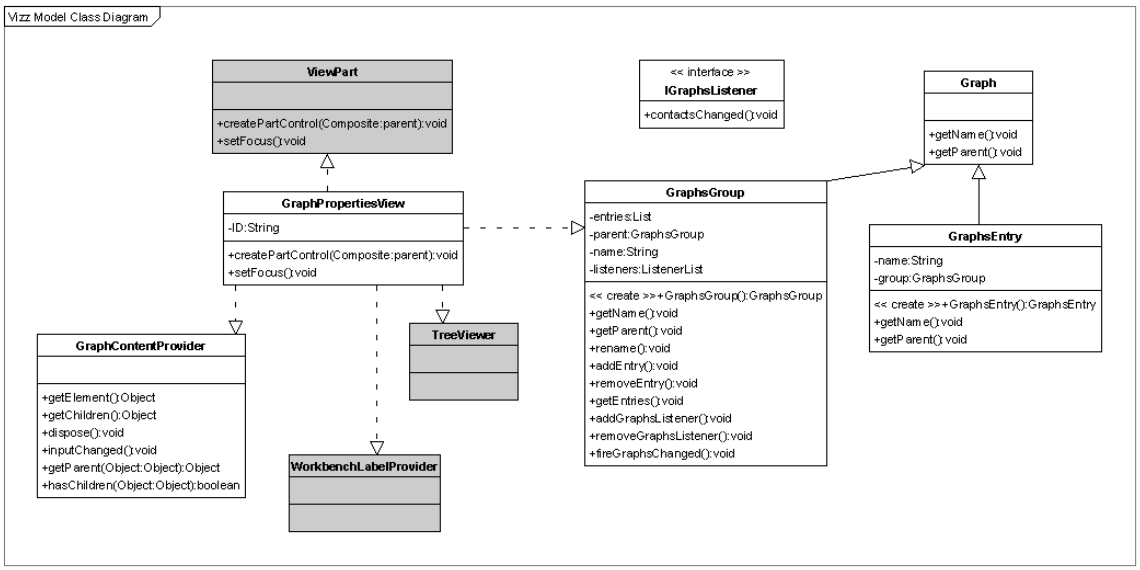

Figure 5.9: Graph Model Class Diagram

Tree Viewers do just what their name implies – display tree structures. Since tree viewers are provided by JFace a content provider and label provider needs be implemented. This is done by the class GraphContentProvider, as seen in Figure 5.9, and in more detail in Figure 5.10. Content providers deliver the tree nodes (e.g., parents and children), and the label provider implemented by

WorkbenchLabelProvider produces human-readable names and representative images for the nodes.

Since the *content provider* resembles quite a central part, we want to look a bit closer on what it does, by discussing some implementation details. Figure 5.10: Graph content provider shows the GraphContentProvider class implementing *ITreeContentProvider* as used in the new VizzAnalyzer. The essential parts are:

getChildren() method is implemented to find the first level of elements to display. Notice that this means the root input object for the TreeViewer is never displayed, that is because it provides the starting point from which the visible tree is built.

getParent() method is implemented to find the parent of any element.

hasChildren() method returns true if element has a children, false otherwise.

```
public class GraphContentProvider implements ITreeContentProvider {

    public Object[] getElements(Object inputElement) {
        return getChildren(inputElement);
    }

1   public Object[] getChildren(Object parent) {
        if (parent instanceof GraphsGroup) {
            return ((GraphsGroup) parent).getEntries();
        }
        return new Object[0];
    }

    public void dispose() {
    }

    public void inputChanged(Viewer viewer, Object oldInput,
            Object newInput) {
    }

2   public Object getParent(Object element) {
        if (element instanceof GraphsGroup) {
            return ((GraphsGroup) element).getParent();
        }
        return null;
    }

3   public boolean hasChildren(Object element) {
        if (element instanceof GraphsGroup)
            return (((GraphsGroup) element).getEntries().length == 0);
        return false;
    }
}
```

Figure 5.10: Graph content provider

```
public class GraphView extends ViewPart {

    public static final String ID =
                "se.vxu.msi.vizzanalyzer.view.graphView";

1   private TreeViewer treeViewer;
    private GraphsGroup graphGroup;
    private ProjectManager pm;
    private IAdapterFactory adapterFactory = new
                        VizzAnalyzerAdapterFactory();

    public GraphView() {
        super();
    }

    public void createPartControl(Composite parent) {
        pm = new ProjectManager();
2       graphGroup = pm.initGraphs();
        treeViewer = new TreeViewer(parent, SWT.MULTI |
                    SWT.H_SCROLL
                    | SWT.V_SCROLL | SWT.BORDER);
        getSite().setSelectionProvider(treeViewer);
        Platform.getAdapterManager().registerAdapters(adapterFactory,
                    Graph.class);
        treeViewer.setLabelProvider(new WorkbenchLabelProvider());
3       treeViewer.setContentProvider(new GraphContentProvider());
        treeViewer.setInput(graphGroup);
        treeViewer.expandAll();
        graphGroup.addGraphsListener(new IGraphsListener() {
            public void contactsChanged(GraphsGroup contacts,
                    GraphsEntry entry) {
                treeViewer.refresh();
                treeViewer.expandAll();
            }
        });
    }

    public void dispose() {
        Platform.getAdapterManager().unregisterAdapters(adapterFactory);
        super.dispose();
    }

    public void setFocus() {
        treeViewer.getControl().setFocus();
    }
}
```

Figure 5.11: Graph View with tree viewer

Figure 5.11: Graph View with tree viewer shows the implementation of the tree view component and the initialization with the previously discussed content provider.

A *TreeViewer* object is created to support the Graph model.

GraphsGroup is initialized by *Data* and *View* and they no have children until the user load any graph.

Initialization of tree viewer with the *GraphContentProvider*.

Figure 5.12: GraphView tree viewer shows how it looks like if everything is put together. A tree viewer with four children: Data, PackageGraph:a, PackageGraph:a, and View. Whereas the two PackageGraph:a elements are children of Data.

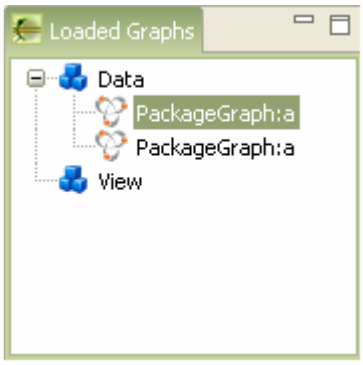

Figure 5.12: GraphView tree viewer

5.5 VizzAnalyzer views

As for the tree view, another central part of the old and new VizzAnalyzer GUI are views. A view displays information about a graph selected in the tree view, displaying the graph properties and properties of its nodes and edges.. The Eclipse Platform provides the concept of editors, which are meant for the primary focus of attention while views usually provide supporting information for a given task. Since the VizzAnalyzer has currently no use for an Editor, we are focusing on the concept of view to display the graph, node and edge information. Therefore information is loaded from graphs as it is described in, and displayed in five independent views, which are Graph View, Node View and Edge View, Graph Properties View and an extra view for the second perspective implemented that contains all the information in the same view. Figure 5.13: Views Class diagram depicts the design of the views as they are implemented in VizzAnalyzer 2.0.

The white ViewPart abstract class is a base implementation of all workbench windows. This class is provided by Eclipse and not from SWT. The white ISelectionListener interface is provided by Eclipse for listening to selection changes.

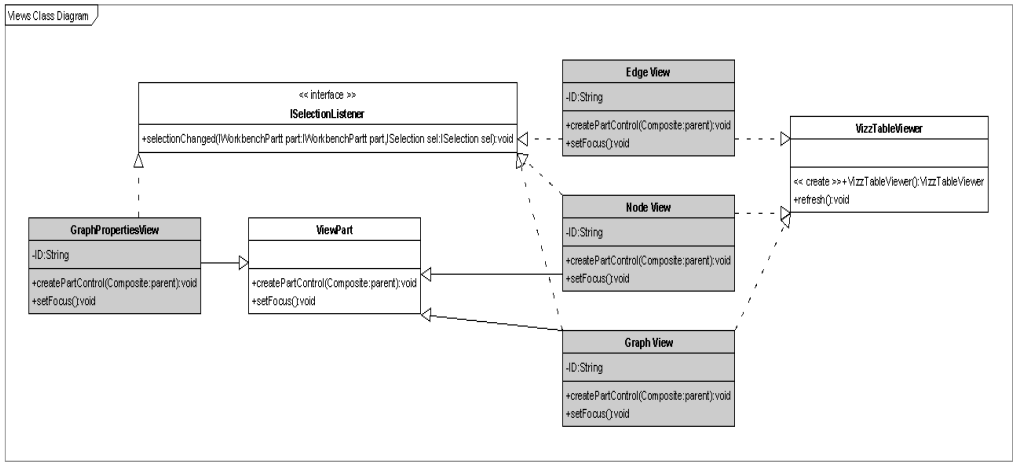

Figure 5.13: Views Class diagram

The gray classes are the views implemented for the VizzAnalyzer 2.0; they need to extend from the abstract ViewPart class and must implement the *createPartControl* method which defines details on the appearance of the view and actions it provides. The view receives information about changes over certain events. E.g., if the user has selected a graph in the tree view, the views need to show the related information. Therefore, the views need to implement the *ISelectionListener* interface in order to be able to react on events. All the views in VizzAnalyzer 2.0 implement the *ISelectionListener* and are therefore aware when the user select a graph on the Graph View implemented.

Each view has an *ID* attribute as unique identification. Using this attribute, it is possible to access a view from any point of the application. Views need to be published in the *plugin.xml* of the application; otherwise they will not be visible. Refer to section.4.1.1 for details.

The *VizzTableViewer* object is instantiate by node, edge and graph information view and it is responsible to refresh the information for each view depending on the parameter received.

Sample code of a view can be seen in Figure 5.14: Sample view implementation. All code had to be implemented from scratch, since it was not possible to reuse the Swing code from the old VizzAnalyzer. Yet, what is till the same is the use of the GRAIL library for working with the graphs. In particular the following is happening:

An ISelectionListener is created for the view and a *selectionChanged* method is implemented. During the life of the application many listeners are invoked with different events. In this case, we are only interested in events of type GraphEntry or GraphGroup.

The listener filters the events and compares the object with a GraphEntry, if the instance are the same means the user has selected a graph and wants to see the information from the graph. The *viewer* object refreshes the view and shows the information on that view.

In case, the user selects a GraphsGroup, the view is disposed because no graph is selected.

The *createPartControl* method creates the viewer and initializes it. This method is only called when the view is activated first time, that means only once during the entire Workbench cycle.

```
Public class GraphPropertiesView extends ViewPart {
    public static final String ID = "se.vxu.msi.vizzanalyzer.view.graphPropertiesView";
    private VizzTableViewer viewer;

1   ISelectionListener listener = new ISelectionListener() {
            public void selectionChanged(IWorkbenchPart part,
                    ISelection sel) {
                if (!(sel instanceof IStructuredSelection))
                    return;
                IStructuredSelection ss = (IStructuredSelection) sel;
                Object o = ss.getFirstElement();
2               if (o instanceof GraphsEntry) {
                    GraphsEntry graphSelected = (GraphsEntry) o;
                    if (graphSelected.getObject() instanceof
                SetBasedDirectedGraph) {
                        SetBasedDirectedGraph graph =
                (SetBasedDirectedGraph) graphSelected
                                .getObject();
```

```
                                        viewer.refresh((GraphInterface) graph,
                                viewer.INFO_TABLE);
                                }
3                       } else if (o instanceof GraphsGroup) {
                                viewer.dispose();
                        }
                }
        };

        /**
         * This is a callback that will allow us to create the viewer and
         Initialize it.
         */
4       public void createPartControl(Composite parent) {
                getSite().getPage().addSelectionListener(listener);
                viewer = new VizzTableViewer(parent);
        }
}
```

Figure 5.14: Sample view implementation

There is a big difference between the VizzAnalyzer 1.0 and 2.0 implementation. VizzAnalyzer 1.0 shows the graph properties, nodes and edges information using a JPanel, having two fixed JTables for displaying the node and edge information. No different views or perspectives are implemented.

The VizzAnalyzer 2.0 uses different views, thus being more flexible than the JPanel approach in version 1.0. These views can be grouped in and shared between perspectives, which improve the flexibility and value of the user interface.

Figure 5.15: Graph properties view result depicts a sample of a view. The General Graph Information tab shows the information associated to the graph when the user selects a graph. Node and Edge Information tab are other views displaying information related for each tab.

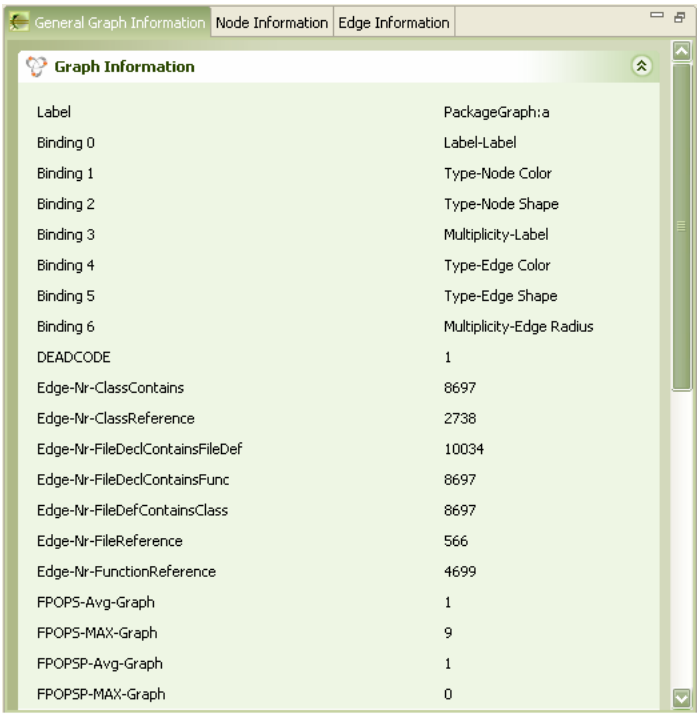

Figure 5.15: Graph properties view result

5.6 Help manager

This section describes how the Eclipse help system is used to provide online help for the VizzAnalyzer. Unlike the features used so far, the help subsystem and its libraries are not included in the RCP SDK, they are a separate feature.

Using the help system requires no code. It is completely configured by a wizard using the VizzAnalyzer product editor. The content of the help system is provided in form of HTML pages. The relevant pages are selected in the wizard, which generates automatically a searchable index and compiles the help system. The help system is reachable anytime over the help menu. In the current version no context sensitive help, where the user navigates to the correct help entry by simply pressing F1 in any place of the application, will be provided. This is left open for future work. Also the generation of useful help content is not part of the thesis, which shall just prepare the ground, allowing the authors of the VizzAnalyzer to concentrate on the content creation.

More detailed information on how to implement a help system using the Eclipse help system can be found in [3]. A brief overview about how to add content to the help system is proved in the appendix A.2.1 Updating help content.

To use the Eclipse help system, making it part of the VizzAnalyzer 2.0 application, it was necessary to add a number of libraries, which are listed in Figure 5.16.

```
org.apache.lucene_1.4.103.v20060601
org.eclipse.help.webapp_3.2.1.R321_v20060803
org.eclipse.tomcat_4.1.130.v20060601
org.eclipse.help.appserver_3.1.100.v20060602
org.eclipse.help.base_3.2.1.R321_v20060822
org.eclipse.help.ui_3.2.0.v20060602
org.eclipse.ui.forms_3.2.0.v20060602
```

Figure 5.16: List help manager dependencies

To inform the Eclipse Framework that there is a help system provide, it is necessary to register the dependencies between the plug-ins, and to register the action on VizzAnalzyerActionBarAdvisor, which is done by the code shown in Figure 5.17, performing these actions:

Since Eclipse provides an Action Factory from its own platform and the VizzAnalyzer implements its help system, it is necessary to call the Eclipse Action Factory registered as *HELP_CONTENTS*.

Add a help menu item to the menu, and associate it with the action loading the help system.

```
/** creating the action starting/displaying the help system**/
1 helpAction = ActionFactory.HELP_CONTENTS.create(window);
  register(helpAction);

/** Adding the action into the Help menu. **/
2 MenuManager helpMenu = new MenuManager("&Help", "help");
    helpMenu.add(helpAction);
```

Figure 5.17: Updating action for help manager

5.7 Update manager

To implement this feature in the VizzAnalyzer some more dependencies have to be satisfied. Some of them were already needed for using the help manager (see 5.6), some others still need to be added to complete the update manager implementation.

The additional plug-ins are described in the following list:

```
/** already used for help manager */
org.eclipse.ui.forms_3.2.0.v20060602

/** new plug-ins added to the list **/
org.eclipse.update.core_3.2.1.v20092006
org.eclipse.update.scheduler_3.2.1.v20092006
org.eclipse.update.ui_3.2.1.v20092006
```

For adding update functionality to the new VizzAnalyzer, it was necessary to define a Eclipse *feature*, or just *feature*. A *feature* collects sets of plug-ins that belong together in order to form some coherent unit of function. *Features* have a very simple structure; they are a list of plug-ins and other features. They comprise a list of "materials" for a set of functions.

Figure 5.18: List plug-ins for the update feature shows the list of plug-ins which comprise the update *feature.* And again, Eclipse provides a wizard for organizing and managing the required plug-ins. As a result, the VizzAnalyzer update feature project is created and opened in a feature editor, Figure 5.19.

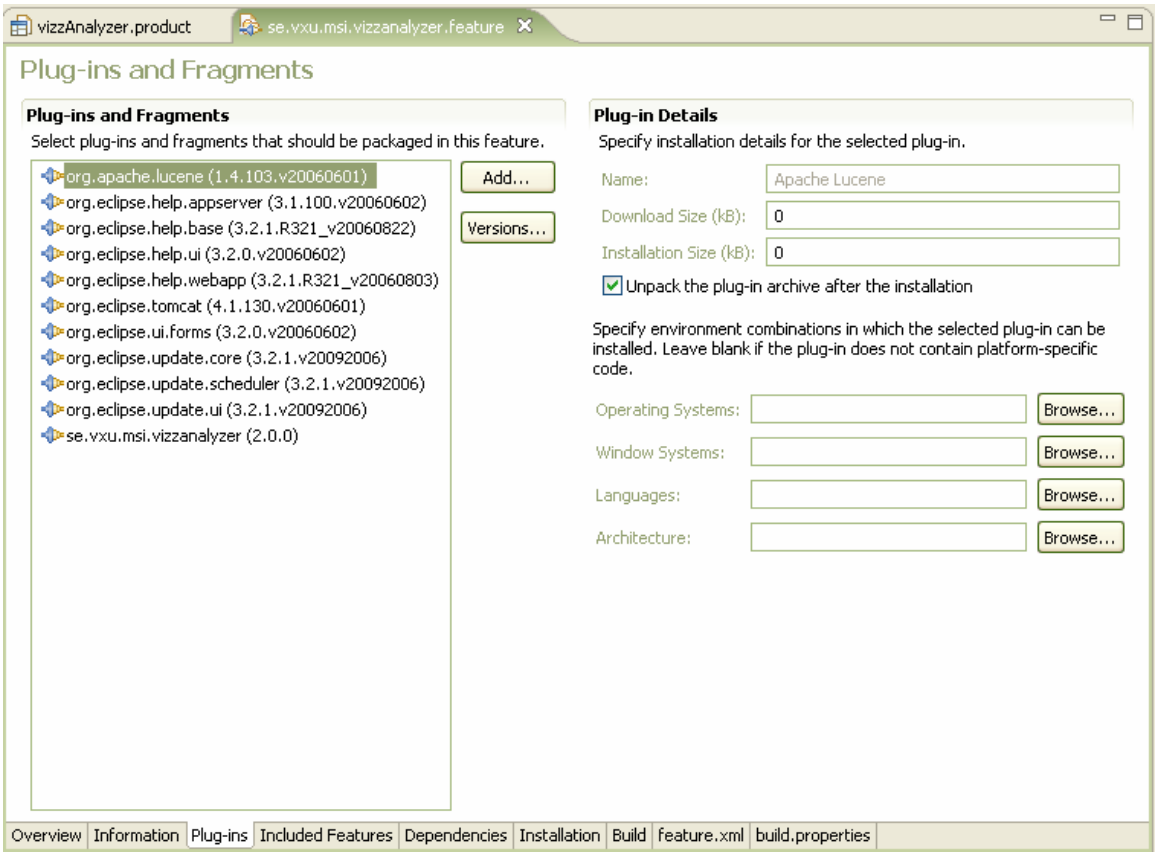

Figure 5.18: List plug-ins for the update feature

The Overview page of the editor shows a General Information section, see Figure 5.19: Adding update site information, showing the values that were entered in the wizard. The *Update Site* information must provide the URL of the update site, where the new versions of the VizzAnalyzer and its components are located. During the course of the thesis, the update site is http://student.msi.vxu.se:3458/users/ddeex05/vizzanalyzer/update, which has been used for testing. This URL needs to be updated once VizzAnalyzer 2.0 is productive.

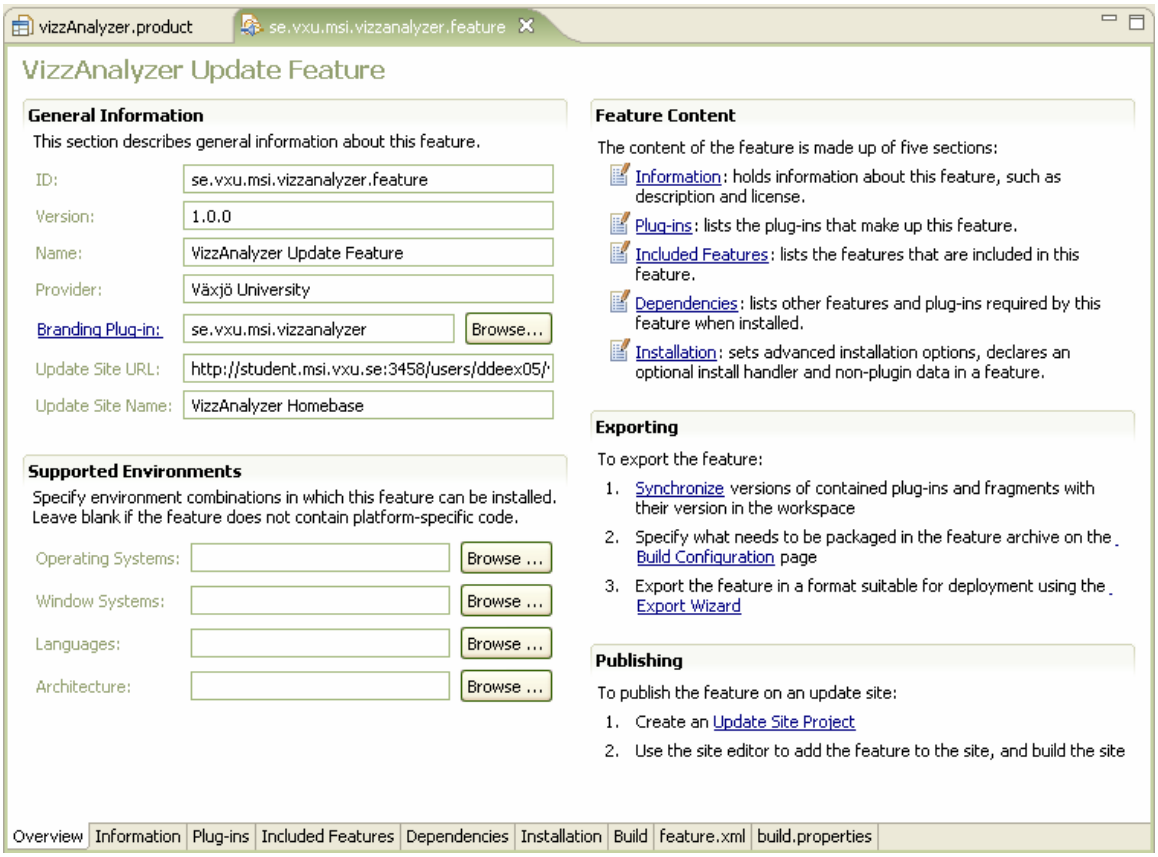
vizzAnalyzer.product
se.vxu.msi.vizzanalyzer.feature
VizzAnalyzer Update Feature
General Information
This section describes general information about this feature.
ID:
se.vxu.msi.vizzanalyzer.feature
Version:
1.0.0
Name:
VizzAnalyzer Update Feature
Provider:
Växjö University
Branding Plug-in:
se.vxu.msi.vizzanalyzer
Browse...
Update Site URL:
http://student.msi.vxu.se:3458/users/ddeex05/
Update Site Name:
VizzAnalyzer Homebase
Supported Environments
Specify environment combinations in which this feature can be installed.
Leave blank if the feature does not contain platform-specific code.
Operating Systems:
Window Systems:
Languages:
Architecture:
Browse ...
Feature Content
The content of the feature is made up of five sections:
Information: holds information about this feature, such as description and license.
Plug-ins: lists the plug-ins that make up this feature.
Included Features: lists the features that are included in this feature.
Dependencies: lists other features and plug-ins required by this feature when installed.
Installation: sets advanced installation options, declares an optional install handler and non-plugin data in a feature.
Exporting
To export the feature:
1. Synchronize versions of contained plug-ins and fragments with their version in the workspace
2. Specify what needs to be packaged in the feature archive on the Build Configuration page
3. Export the feature in a format suitable for deployment using the Export Wizard
Publishing
To publish the feature on an update site:
1. Create an Update Site Project
2. Use the site editor to add the feature to the site, and build the site
Overview
Information
Plug-ins
Included Features
Dependencies
Installation
Build
feature.xml
build.properties

Figure 5.19: Adding update site information

There are three main functions supported by Update:

Searching for updates to existing features – This finds and installs updates to existing features. For example, if you are running VizzAnalyzer 2.0 and 2.1 is released, the update procedure will allow the users to install the new version of VizzAnalyzer, thereby replacing the old version. Figure 5.20: Updating action implementation shows the code for this, performing the following actions:

The constructor sets the main properties for the Update action.

A busy indicator starts executing the UpdateJob.

UpdateManagerUI opens the installer of the Update Manager and starts the job, in this case, connect to the web and check if a new VizzAnalyzer version is available.

```
public class UpdateAction extends Action implements IAction {
    private IWorkbenchWindow window;

1   public UpdateAction(IWorkbenchWindow window) {
        this.window = window;
        setId("se.vxu.msi.vizzanalyzer.newUpdates");
        setText("&Update...");
```

```
            setToolTipText("Search for updates to VizzAnalyzer");
            setImageDescriptor(AbstractUIPlugin.imageDescriptorFromPlugin(
                        VizzAnalyzerApplication.PLUGIN_ID,
                       "icons/uSearchObj.gif"));
            window.getWorkbench().getHelpSystem().setHelp(this,
                        "se.vxu.msi.vizzanalyzer.updates");
    }

    public void run() {
2       BusyIndicator.showWhile(window.getShell().getDisplay(),
                        new Runnable() {
                    public void run() {
3                     UpdateJob job = new UpdateJob("Searching for
                        updates", false,
                                                        false);
                 UpdateManagerUI.openInstaller(window.getShell(),
                        job);
                    }
            });
    }
}
```

Figure 5.20: Updating action implementation

Searching for new features – VizzAnalyzer could be extended with other features making it more extensible. Therefore it allows users to to search for new features, such as future plug-ins from the VizzAnalyzer Framework, like new metrics, front-ends, or other VizzAnalyzer or Eclipse plug-ins.

The *AddExtensionAction* was added in the same way as the UpdateAction, Figure 5.21. This action searches new features the VizzAnalyzer currently does not have in the following steps:

The constructor of AddExtensionAction setting the main properties.

A *getSearchRequest()* method is called looking for new features.

The *getSearchRequest()* method implementation, the method search request for features that are not already installed, compatible with the current code (BackLevelFilter) and environment (EnvironmentFilter).

```
public class AddExtensionAction extends Action implements IAction {
    private IWorkbenchWindow window;

1   public AddExtensionAction(IWorkbenchWindow window) {
            this.window = window;
            setId("se.vxu.msi.vizzanalyzer.newExtensions");
            setText("&Add Extensions...");
            setToolTipText("Search for new extensions  for
                VizzAnalyzer");
            setImageDescriptor(AbstractUIPlugin.imageDescriptorFromPlugin(
                        VizzAnalyzerApplication.PLUGIN_ID,
                "icons/uSearchObj.gif"));
                window.getWorkbench().getHelpSystem().setHelp(this,
                        "se.vxu.msi.vizzanalyzer.updates");
    }

    public void run() {
```

```
                BusyIndicator.showWhile(window.getShell().getDisplay(),
                        new Runnable() {
                    public void run() {
                            UpdateJob job = new UpdateJob("Search for new
                                extensions",
2                                                       getSearchRequest());

                UpdateManagerUI.openInstaller(window.getShell(),
                                job);
                        }
                });
        }

3       private UpdateSearchRequest getSearchRequest() {
                UpdateSearchRequest result = new UpdateSearchRequest(
                                UpdateSearchRequest.createDefaultSiteSearchCategory(),
                                new UpdateSearchScope());
                result.addFilter(new BackLevelFilter());
                result.addFilter(new EnvironmentFilter());
                UpdateSearchScope scope = new UpdateSearchScope();
                try {
                        String homeBase = System
                                        .getProperty("vizzAnalyzer.homebase",

            "http://student.msi.vxu.se:3458/users/ddeex05/vizzanalyzer/update");
                        URL url = new URL(homeBase);
                        scope.addSearchSite("VizzAnalyzer site", url, null);
                } catch (MalformedURLException e) {
                        // skip bad URLs
                }
                result.setScope(scope);
                return result;
        }
}
```

Figure 5.21: AddExtension action implementation

Managing, the existing configuration of features – Users can enable/disable or remove previously installed features. The necessary code is shown in Figure 5.22, and consists out of these steps:

The constructor as the others actions implemented setting the main properties for the action.

Open the configuration manager from the Update manager UI based on Eclipse.

```
public class ManageExtensionAction extends Action implements IAction {
    private IWorkbenchWindow window;

1   public ManageExtensionAction(IWorkbenchWindow window) {
            this.window = window;
            setId("se.vxu.msi.vizzanalyzer.manageExtensions");
            setText("&Manage Extensions...");
            setToolTipText("Manage extensions  to VizzAnalyzer");
            setImageDescriptor(AbstractUIPlugin.imageDescriptorFromPlugin(
                            VizzAnalyzerApplication.PLUGIN_ID,
              "icons/uConfigs.gif"));
```

```
                window.getWorkbench().getHelpSystem().setHelp(this,
                                "se.vxu.msi.vizzanalyzer.updates");
        }



        public void run() {
                BusyIndicator.showWhile(window.getShell().getDisplay(),
                        new Runnable() {
                        public void run() {
                  UpdateManagerUI.openConfigurationManager(
                                        window.getShell());
                        }
                });
        }
}
```

Figure 5.22: Manage Extension action implementation

5.8 Console

As discussed before, VizzAnalyzer 1.0 provides output to the system console using the Java output streams System.out and System.err. It further provides selected output to an internal console window. Using the Eclipse Framework, the system console and the internal console window is not available any longer and all output of the VizzAnalyzer and its plug-ins need to be redirected to the Eclipse MessageConsole.

From this follows that future applications being part of the VizzAnalyzer framework or integrated plug-ins are using this MessageConsole for textual console output.

A console has been implemented for VizzAnalyzer 2.0. The console extends from *MessageConsole* using the same type of console than Eclipse Platform uses. The core of its implementation is provided in Figure 5.23.

```
/**
 *
 * How to use it (sample):
        ConsoleTab.println(ConsoleTab.ID,
                        "Hello Console",
                        ConsoleTab.MSG_INFORMATION);
 *
 * @author David Ruiz de Azua Nieto
 * @see println method
 */

public class ConsoleTab extends MessageConsole {
        public static final String ID = "VizzAnalyzer Console";
        private static MessageConsole fMessageConsole = null;
        public static final int MSG_INFORMATION = 1;
        public static final int MSG_ERROR = 2;
        public static final int MSG_WARNING = 3;

        public ConsoleTab() {
                super(ID, null);
        }
```

```
        private static MessageConsole getMessageConsole() {
                return fMessageConsole;
        }

1       private static MessageConsole findConsole(String name) {
                ConsolePlugin plugin = ConsolePlugin.getDefault();
                IConsoleManager conMan = plugin.getConsoleManager();
                IConsole[] existing = conMan.getConsoles();
                for (int i = 0; i < existing.length; i++)
                        if (name.equals(existing[i].getName()))
                                return (MessageConsole) existing[i];

                // no console found, so create a new one
                MessageConsole myConsole = new MessageConsole(name, null);
                conMan.addConsoles(new IConsole[] { myConsole });
                return myConsole;
        }

2       private static MessageConsoleStream
                        getNewMessageConsoleStream(int msgKind) {

                int swtColorId = SWT.COLOR_DARK_GREEN;

                switch (msgKind) {
                case MSG_INFORMATION:
                        swtColorId = SWT.COLOR_DARK_GREEN;
                        break;
                case MSG_ERROR:
                        swtColorId = SWT.COLOR_DARK_MAGENTA;
                        break;
                case MSG_WARNING:
                        swtColorId = SWT.COLOR_DARK_BLUE;
                        break;
                default:
                }

                MessageConsoleStream msgConsoleStream =
                getMessageConsole().newMessageStream();

          msgConsoleStream.setColor(Display.getCurrent().
                          getSystemColor(swtColorId));
                return msgConsoleStream;
        }
        /**
         * @param consoleName
         *            ConsoleTab.ID is the default console for VizzAnalyzer and where is show
the messages up.
         * @param msg
         *            Message to write on the console.
         * @param msgKind
         *            Type of message, INFORMATION,
                                ERROR or
                                WARNING type available.*/
3       public static void println(String consoleName,
```

```
                    String msg,
                    int msgKind) {
                fMessageConsole = findConsole(ID);
                getNewMessageConsoleStream(msgKind).println(msg);
        }
    }
```

Figure 5.23: Console implementation

findConsole method check if the console from VizzAnalyzer is activated, if not it creates a new one.

getMessageConsoleStream method is responsible to generate the output, it depends on the type of message, the console shows a different color. Different options are possible as a warning, error or information message.

The *println* method is responsible to print out the message to the VizzAnalyzer console.

Figure 5.24: Screenshot VizzAnalyzer console depicts how the console shows the messages in the VizzAnalyzer console.

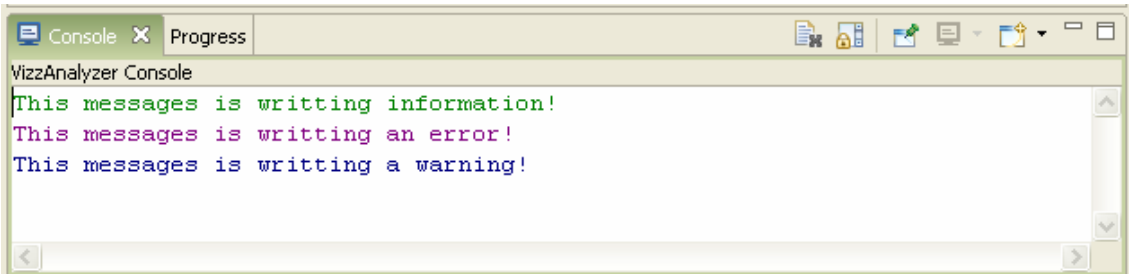

Figure 5.24: Screenshot VizzAnalyzer console

Additional information for VizzAnalyzer developers is provided in the appendix, A.2.4 How to use the VizzAnalyzer console.

5.9 Loading dynamic plug-ins

The most important part of the VizzAnalyzer Framework its ability to load dynamic plug-ins, which are connected to the hot-spots (cf. 4.2), making them available through the menu bar. Refactoring this feature of the old VizzAnalyzer transferring it into VizzAnalyzer 2.0 was also the most difficult part since many different applications are evolved in this process.

Figure 5.25: VizzAnalyzer 1.0 loading dynamic plug-ins depicts how the VizzAnalyzer 1.0 is loading the dynamic plug-ins. For loading the dynamic plug-ins, the VizzAnalyzer needs information from the Tiny-XML Editor (frozen-spot, cf. 4.2), describing where the dynamic plug-ins are. The Tiny-XML editor gets this information from configuration files provided as in XML format. The VizzAnalyzer is building the menus according to the classes resembling the plug-ins. It associates these plug-in classes with the actions in the menu. Thus the plug-ins are accessible through the menu, and can be activated. Figure 5.1: GUI design from VizzAnalyzer 1.0 depicts all the sequence more detailed.

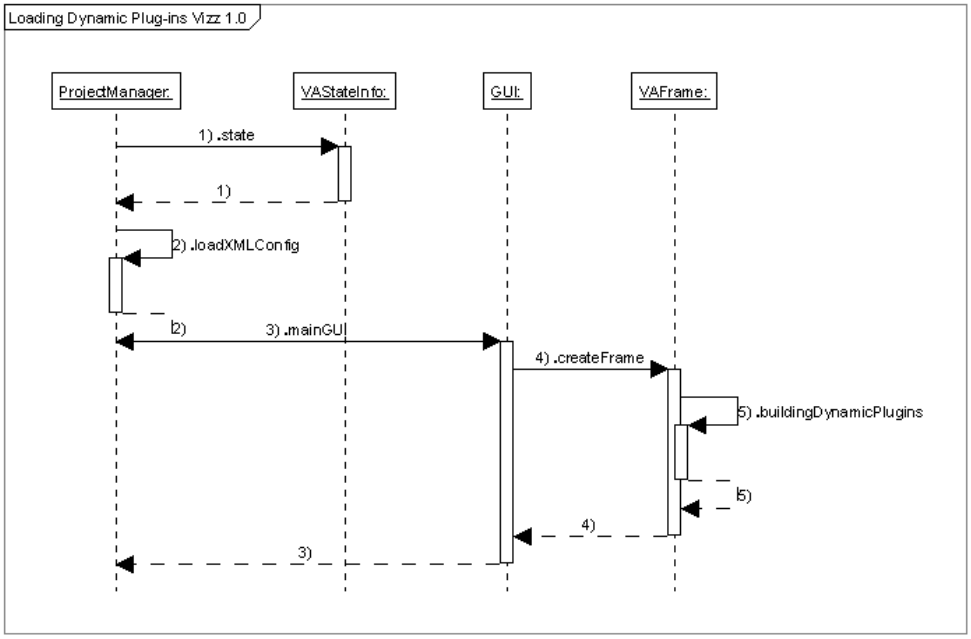

Figure 5.25: VizzAnalyzer 1.0 loading dynamic plug-ins

Figure 5.25 shows that in the past, there was a direct dependency to the GUI, for the new VizzAnalyzer there is less dependencies with the GUI. Figure 5.26: VizzAnalyzer 2.0 loading dynamic plug-ins depicts how the *VizzAnalyzerActionBarAdvisor* is responsible to load the plug-ins when has read the information from *loadXMLEditor.*

The methods executed on *VAFrame* class have been moved to *VizzAnalyzerActionBarAdvisor* since the main frame is built on that class.

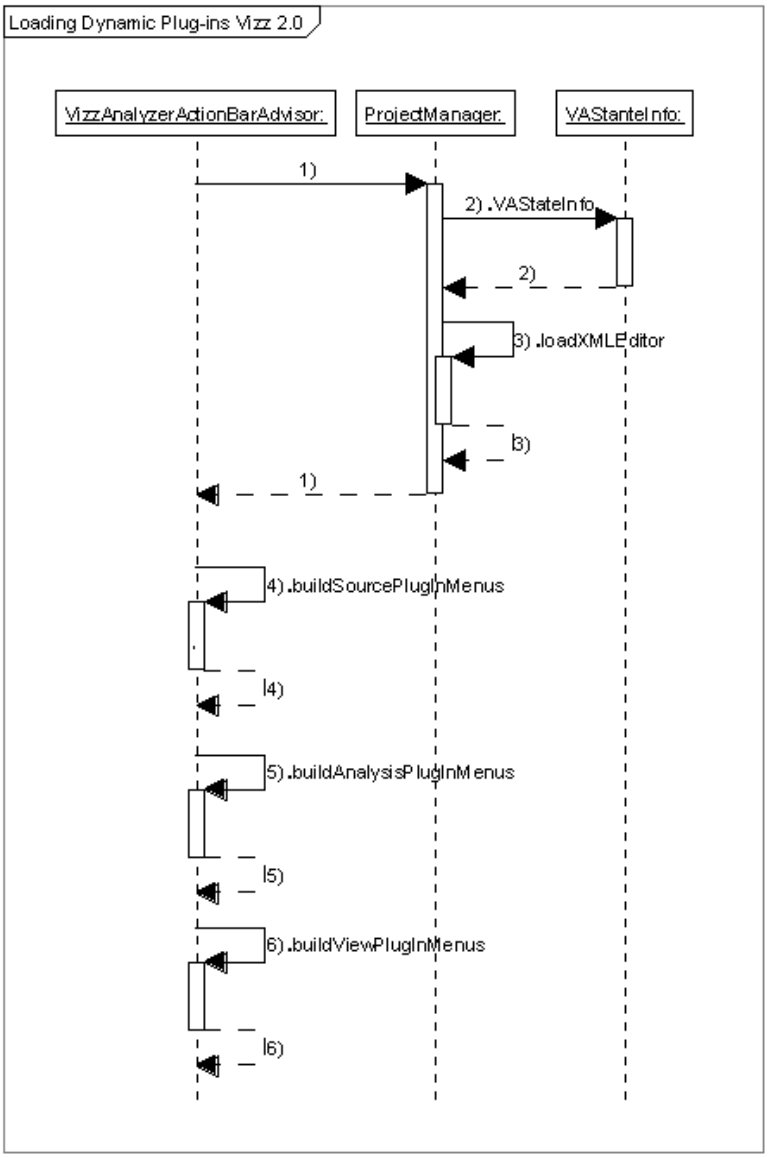

Figure 5.26: VizzAnalyzer 2.0 loading dynamic plug-ins

All dynamic plug-ins loaded by the VizzAnalyzer extend from the *PluginInteface* interface and they return a set of Swing actions. The new VizzAnalyzer accepts those actions by wrapping them into SWT actions. In future the plug-ins should be updated to provide JFace/SWT actions, which is not topic of this thesis.

We limit us in the context of this thesis, to providing and test just one plug-in of each category, which are for the retrieval plug-ins, the Recoder Plug-in, for the analysis plug-in a few metrics, and for the visualization plug-ins, the Vizz3D plug-in.

6 Conclusions and Future Work

This last chapter reflects on the results of the thesis. It points out the extent to which the problem described in the introduction has been solved and if all the goals have been met and the criteria fulfilled. It further discusses possibilities and ideas as to how the outcome of the thesis could be improved, and directions of future developments.

6.1 Conclusions

We described in the past chapters of this thesis a complex refactoring and reengineering task. We started in the introduction by explaining the problem to solve and the motivation for it. As previously stated, the problem addressed by this thesis is:

"Port the VizzAnalyzer Framework from Swing architecture to SWT architecture into the Eclipse Platform offering the same functionality as the VizzAnalyzer version 1.0 GUI".

The reason making this was necessary, was to provide a improved look & feel for the users of the VizzAnalyzer, thereby increasing its acceptance, and to lay the foundations for introducing advanced technology into the VizzAnalyzer, e.g., Eclipse Java and C++ parsers, update features, or help systems.

We presented the goals for solving this problem Section 1.3, together with appropriate criteria. The result of our work can be seen in Figure 6.2, showing the new VizzAnalyzer 2.0, in contrast to this, the old VizzAnalyzer 1.0, can be seen in Figure 6.1.

The individual goals resulting from the problem statement have been addressed as following.

The first goal was to reverse engineer the VizzAnalyzer 1.0, in order to get a good understanding of its architecture and implementation. This goal was met by documenting the architecture of the old VizzAnalyzer and the Eclipse Platform in Chapter 4. From the understanding gained, it was possible to synthesize a new architecture combing the Eclipse Platform and the VizzAnalyzer. This new architecture has been documented in Chapter 4.2.

The second goal was the design and implementation of an alternative VizzAnalyzer GUI allowing for a improved look & feel matching the host platform, thereby providing the same functionality as the original VizzAnalyzer, including basic functionalities as load/save graphs, display graph properties, edge and node information are essential. Furthermore, scripting and recording menus needs to be provided as well as their basic functionality. Additionally, a set of dynamic plug-ins must be loaded on the VizzAnalyzer startup. Analyzer, Visualization and Retrieval menus shall be present after startup. This goal has been met by using the Eclipse Platform as base architecture and implementing, since it uses SWT, which provides naturally a look & feel, which is the same as the operating system it is running on. Furthermore, the same functionality is available for the user being possible to load graphs, save, copy and cut. Displaying the graph, node and edge information was extensively discussed in 5.4 and 5.5. Loading dynamic plug-ins into the new architecture is possible, and was discussed in 5.9. The success in reaching this goal an in particular be shown by comparing the GUI of the old VizzAnalyzer being depicted in Figure 6.1: VizzAnalyzer 1.0 GUI. With the new GUI shown in Figure 6.2: VizzAnalyzer 2.0 GUI.

The third goal was to extend the new VizzAnalyzer using new features from Eclipse platform, which included using a help and update manager, a console and the progress service that Eclipse Platform offers. The solution meeting this criteria has

been implemented and its design been discussed in Sections 5.3, 5.6, 5.7, 5.8 and 5.9. For a better understanding of the requirements, a Use Case analysis has been performed in Chapter 3, on which the architecture and the design and implementation are based on.

The fourth goal is the portability of the VizzAnalyzer, to platforms like Windows, UNIX, and Linux. Reaching this goal did not require any particular action, since the export wizards of the development environment help the developer in accomplishing these tasks. No additional code is needed for adapting the look & feel of the application to the different target platforms.

The fifth and last goal was to improve the performance. This is fulfilled when the GUI related actions are performed faster, or when process information is presented for long activities, e.g., loading graphs. To proof that the first criterion is met is difficult, since there are not proper benchmarks available, testing the speed of GUI components. Yet it is known, that SWT implementations are in general faster, because their architecture works more efficient. The second criterion is met by providing progress information for activities taking more than a few seconds, thereby providing a visual feedback to the user. This has been discussed in A.2.5 How to use the progress service.

The previous paragraphs show that all goals stated in the beginning of this thesis have been successfully addressed. Therefore it can be concluded that the problem underlying this thesis has been solved.

Additionally, by choosing to implement the VizzAnalyzer as plug-in into the Eclipse Platform, it was possible to add advanced functionalities like help or update manager, progress service, console and different perspectives at a rather low effort.

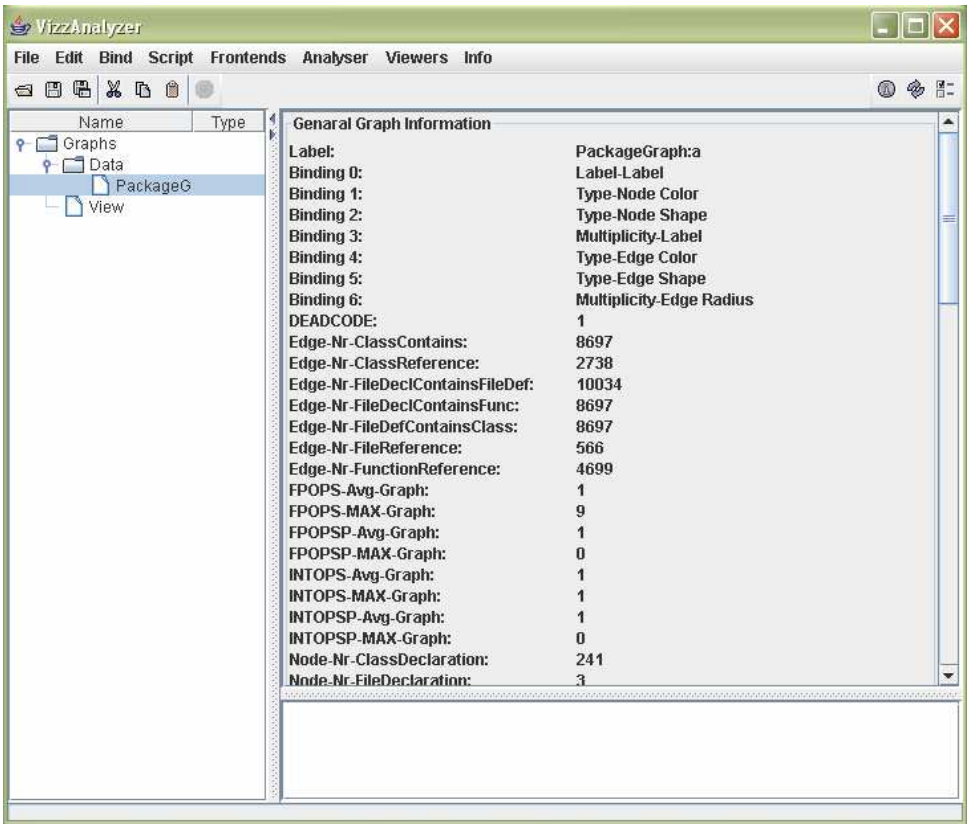

Figure 6.1: VizzAnalyzer 1.0 GUI

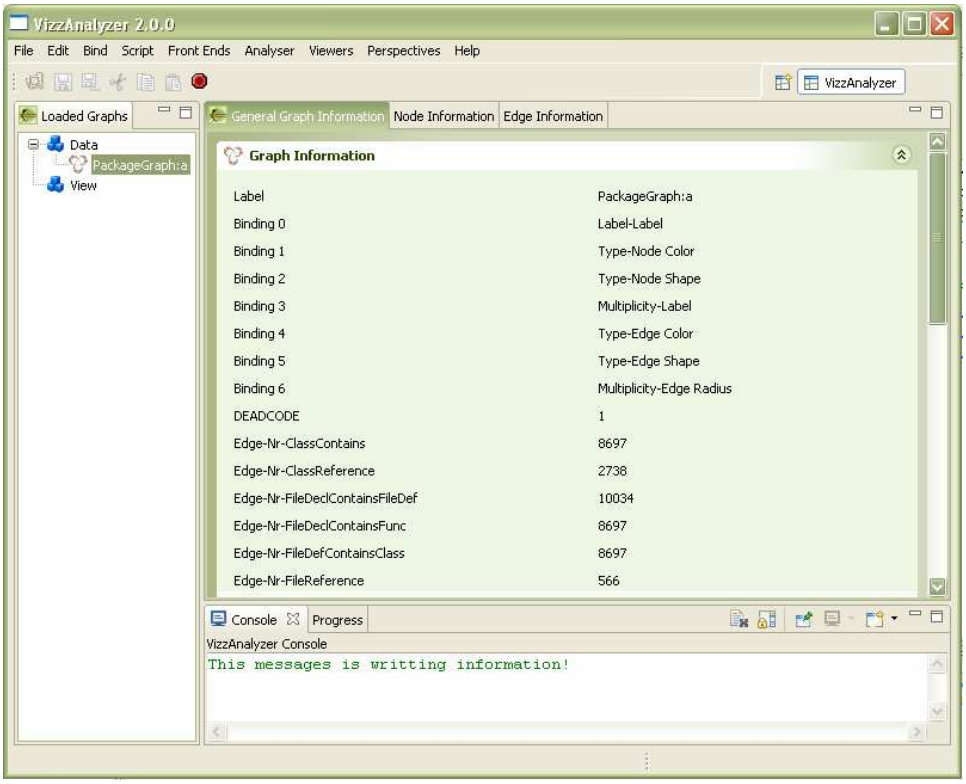

Figure 6.2: VizzAnalyzer 2.0 GUI

In my opinion the topic for this thesis has been very interesting since I could learn myself a lot of specific knowledge about the Eclipse platform, and how to create plug-ins for it. I am sure this knowledge will help me in my future, since more and more companies start to base their applications on the Eclipse platform. Further, I could participate in a big refactoring project where I could use much of the knowledge I learned about so far only in theory, e.g., requirements engineering, analysis and design of complex systems. It was very motivating to know that my efforts will help the authors of the VizzAnalyzer in their efforts to maintain and improve the VizzAnalyzer.

6.2 Future Work

The implementation of VizzAnalyzer 2.0 serves its purpose and it is well suitable for solving the problem addressed by this thesis. Yet during the development certain improvements where noted. These regard mainly parts of the GUI, not belonging to the main user interface, but are dialogs, mainly from plug-ins and utility functions. These have not been adopted SWT, but remained in Swing. These should be successively updated, to provide a consistent GUI completely implemented in SWT. They need to be added as features to VizzAnalyzer 2.0 so that users can download them over the update feature.

Furthermore, guidelines should be published, helping the developers of plug-ins to facilitate the new features of the VizzAnalyzer, integrating with the console, progress, service, help and update features.

Since only few dynamic plug-ins could be refactored for demonstration purposes it is still necessary that the authors of these plug-ins adapt them to integrate seamlessly into the VizzAnalyzer 2.0.

Most of all, a major refactoring is needed of the *PluginInterface*, which should be changed to return SWT actions instead of the Swing actions. This kind of refactoring

was not part of the thesis, since it affects a large number plug-ins, and should be best solved by the authors of these plug-ins, having the necessary in deep knowledge.

Reading the classpath for the Analysis, Retrieval and Visualization plug-ins and loading it to the dynamic menu had some problems. A new approach is described on A.2.6 Class path Configuration describing another way to implement future plug-ins for the VizzAnalyzer 2.0.

References

[1] Thomas Panas, Rüdiger Lincke, Welf Löwe. “The VizzAnalyzer Handbook”. Technical Report. October 2005.Växjö University.

[2] Eclipse. www.eclipse.org.

[3] Jeff McAffer, Jean-Michel Lemieux. “Eclipse Rich Client Platform”. Editorial Addison Wesley. ISBN 0-321-33461-2.

[4] David Ruiz de Azua Nieto. “Integration of a Standard-Based Quality Assessment into the VizzAnalyzer. June 2006. Växjö University.

[5] Rajorshi Biswas. The Eclipse IDE. GNU Free Documentation License, http://freehosting.hostrave.com/p/rajorshi/img/eclipse/part2/fig1.png.

[6] David Gallardo. “Getting start with Eclipse Platform”. November 2002. http://www-128.ibm.com/developerworks/library/os-ecov/

[7] Nick Edgar, Keving Haland, Jin Li and Kimberley Peter “Eclipse User Interface Guidelines Version2.1.” http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.

[8] Yannick Saillet. “Migrate your Swing application to SWT”. DevelopersWorks IBM. January 2004.

[9] Azad Bolour, Bolour. “Notes on the Eclipse Plug-in Architecture”. July 3, 2003 http://www.eclipse.org/articles/Article-Plug-in architecture/plugin_architecture.htm

[10] Koray Güclü. “A First Look at Eclipse Plug-in Programming. http://www.developer.com/java/other/article.php/10936_3316241_2

[11] Progress services for Eclipse 3.0 and concepts. http://wiki.eclipse.org/index.php/FAQ_Actions%2C_commands%2C_operations%2C_jobs:_What_does_it_all_mean%3F#Jobs

About Views and Perspectives

[12] Dave Springgay. "Creating an Eclipse View". November 2, 2001. http://www.eclipse.org/articles/viewArticle/ViewArticle2.html

[13] Eclipsepedia. "FAQ How do I make a view respond to selection changes in another view?". June 9, 2006. http://wiki.eclipse.org/index.php/FAQ_How_do_I_make_a_view_respond_to_selection_changes_in_another_view%3F

[14] Chinmay Pandit, Software Engineer, IBM India Software Labs. "Make your Eclipse applications richer with view linking". http://www-128.ibm.com/developerworks/opensource/library/os-ecllink/?ca=dgr-lnxw03EclipseGUIviews

[15] Anthony Hunter, IBM. "The Eclipse Tabbed Properties View". February 19, 2006. http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html

[16] Dave Springgay, OTI. "Using Perspectives in the Eclipse UI". August 27, 2001. http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html

About Progress dialog and progress view

[17] Eclipsepedia. "FAQ How do I switch from using a Progress dialog to the Progress View?". http://wiki.eclipse.org/index.php/FAQ_How_do_I_switch_from_using_a_Progress_dialog_to_the_Progress_view%3F

[18] Kenneth Ölwing, BEA JRPG. "How to Correctly and Uniformly Use Progress Monitors". January 18, 2006. http://www.eclipse.org/articles/Article-Progress-Monitors/article.html

About Jobs

[19] Michael Valenta, IBM OTI Labs. "On the Job: The Eclipse Jobs API". September 20, 2004. http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html

About Selection Service

[20] Marc R. Hoffmann. "Eclipse Workbench: Using the Selection Service". April 14, 2006. http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html

About SWT (Standard Widget Toolkits)

[21] Carolyn MacLeod, OTI. "Understanding Layouts in SWT". March 22, 2001. http://www.eclipse.org/articles/Article-Understanding-Layouts/Understanding-Layouts.htm

[22] Shantha Ramachandran, Christopher Batty and Weichi Truong. "Advanced SWT Widgets". February 20, 2004. http://www.cs.umanitoba.ca/~eclipse/3-Advanced.pdf

About Tables and Tree Viewer

[23] Chris Grindstaff, Applied Reasoning. "How to use the JFace Tree Viewer". May 5, 2004. http://www.eclipse.org/articles/Article-TreeViewer/TreeViewerArticle.htm

[24] Grant Gayed, IBM Ottawa Lab. "Custom Drawing Table and Trees Items". September 15th, 2006. http://www.eclipse.org/articles/Article-CustomDrawingTableAndTreeItems/customDraw.htm

[25] Laurent Gauthier, Mirasol Op'nWorks. "Building and delivering a table editor with SWT/JFace". July 3, 2003. http://www.eclipse.org/articles/Article-Table-viewer/table_viewer.html

[26] Beatriz Laderoza and Grant Gayed, IBM Ottawa Lab. "Improving your SWT application performance by using a virtual Table or Tree". June 5th, 2006. http://www.eclipse.org/articles/Article-SWT-Virtual/Virtual-in-SWT.html

About Images

[27] Chengdong Li, Research in Computing for Humanities, University of Kentucky. "A Basic Image Viewer". March 15th, 2004. http://www.eclipse.org/articles/Article-Image-Viewer/Image_viewer.html

About Eclipse Overview

[28] Jim Amsden, OTI and Andrew Irvine, OTI. "Your First Plug-in". January 28th, 2003. http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html

[29] Ed Burnette, SAS. "Rich Client Tutorial Part 2". February 6th, 2006. http://www.eclipse.org/articles/Article-RCP-2/tutorial2.html

About Classpath

[30] Eclipsepedia. "FAQ What is the classpath of a plug-in?". http://wiki.eclipse.org/index.php/FAQ_What_is_the_classpath_of_a_plug-in%3F

[31] Eclipsepedia. "FAQ What is the plug-in manifest file (plugin.xml)?". http://wiki.eclipse.org/index.php/FAQ_What_is_the_plug-in_manifest_file_%28plugin.xml%29%3F

[32] Eclipsepedia. "FAQ How do I make my plug-in connect to other plug-ins?". http://wiki.eclipse.org/index.php/FAQ_How_do_I_make_my_plug-in_connect_to_other_plug-ins%3F

[33] Eclipsepedia. "FAQ How do I add a library to the classpath of a plug-in?". http://wiki.eclipse.org/index.php/FAQ_How_do_I_add_a_library_to_the_classpath_of_a_plug-in%3F

[34] Eclipsepedia. "FAQ How can I share a JAR among various plug-ins?". http://wiki.eclipse.org/index.php/FAQ_How_can_I_share_a_JAR_among_various_plug-ins%3F

[35] Eclipsepedia. "FAQ How do I use the context class loader in Eclipse?". http://wiki.eclipse.org/index.php/FAQ_How_do_I_use_the_context_class_loader_in_Eclipse%3F

Appendix A. The VizzAnalyzer 2.0

This section contains two subchapters. The first subchapter is a user manual explaining the new features implemented for the VizzAnalyzer. The second subchapter is a tutorial for developers who want to continue developing the VizzAnalyzer 2.0.

A.1 User Manual

This chapter is intended to assist the end-user. It explains how to use the main functions of VizzAnalyzer 2.0.

A.1.1 Installation and running

This section contains instructions on how to install and run the VizzAnalyzer. The following scenario is running the application in a Windows platform. The application is provided as ZIP file. Section A.2.3 Exporting VizzAnalyzer describes how to export the application for different operating systems.

To install the VizzAnalyzer, simply unzip the ZIP file to an installation folder of your choice. After the installation an *eclipse* folder is created. (See Figure A.1: Installation folder). There is a VizzAnalyzer executable file. Starting that file will run the VizzAnalyzer.

Figure A.1: Installation folder

A.1.2 Product key

During the first start of the VizzAnalyzer 2.0 it asks for a serial number and name for registration purposes. If the serial is wrong the start of the VizzAnalyzer 2.0 is aborted. Note that the dialog is still using Swing, since it is provided by a component of the VizzAnalyzer not belonging to the main GUI. This should be replaced with a SWT implementation.

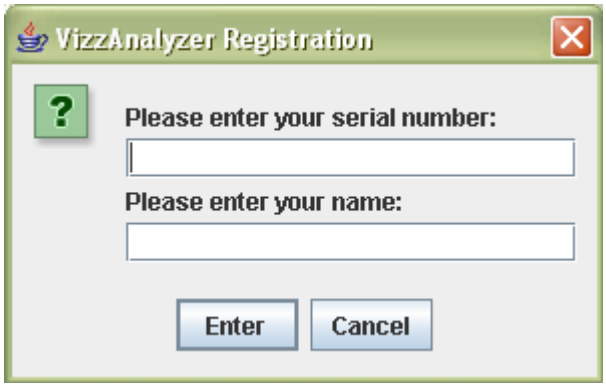

Figure A.2: Registration dialog

After supplying a correct serial number, the main window of the VizzAnalyzer appears, as seen in Figure A.3: Default perspective. It shows the default VizzAnalyzer perspective. The user can interact with VizzAnalyzer 2.0 using the menus. Different views on the graphs loaded are available over tabs. They provide in detail General graph, Node and Edge information. On the left side is the tree view making loaded graphs are accessible. A console and progress view is presented to give information to the user (lower part of Figure A.3).

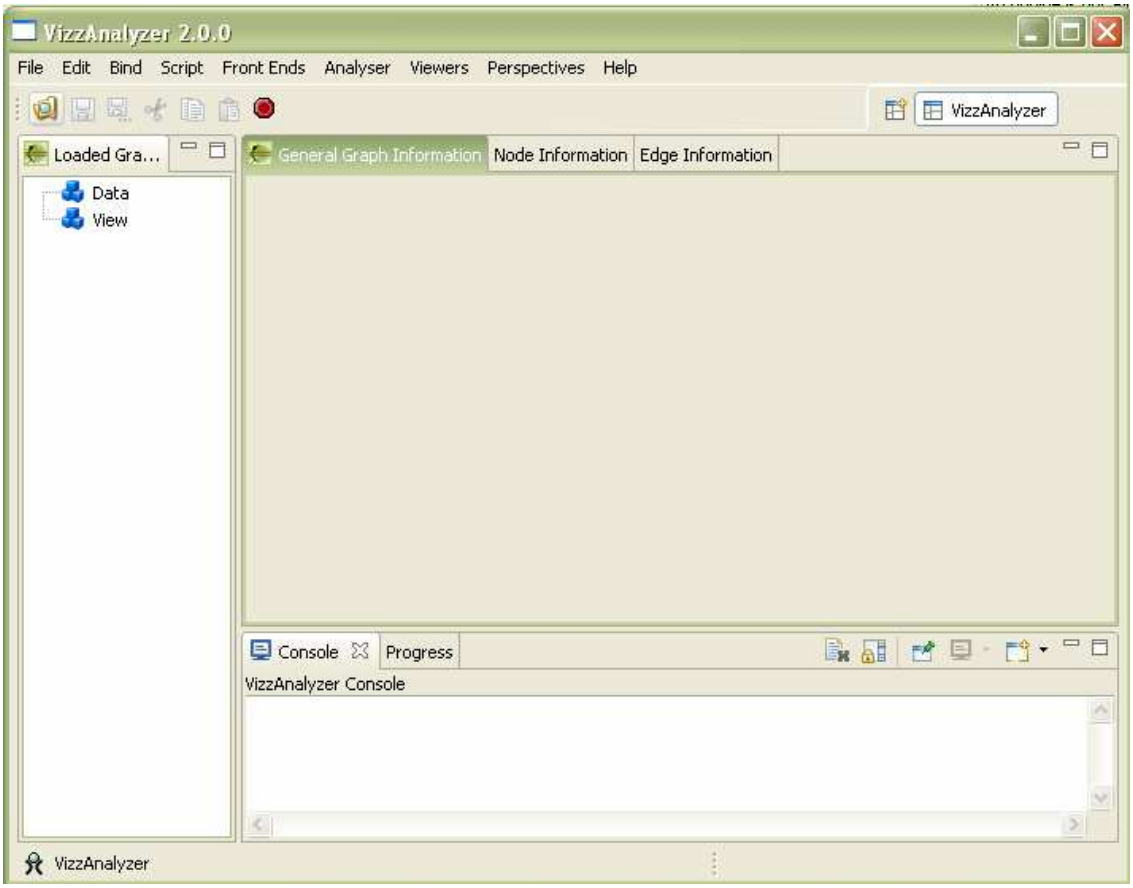

Figure A.3: Default perspective

A.1.3 Loading/saving graphs

Use the File menu or toolbar to load graphs. The *Save* and *Save As…* entry is only enabled, if a graph is loaded.

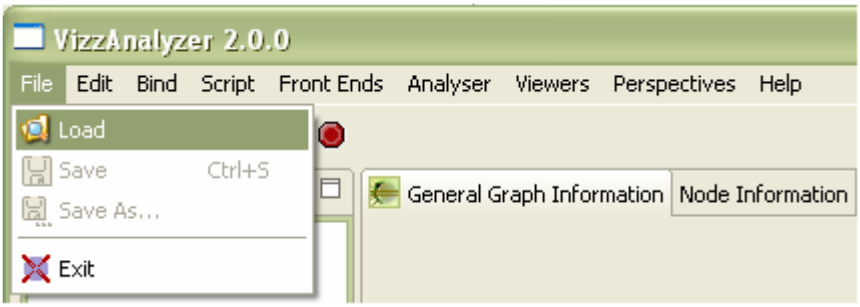

Figure A.4: Loading graphs menu

A dialog is shown where the user can choose a graph file supported by the VizzAnalyzer. In Figure A.5: Loading graphs dialog, the *KOLAH.gml* file has been selected as a sample.

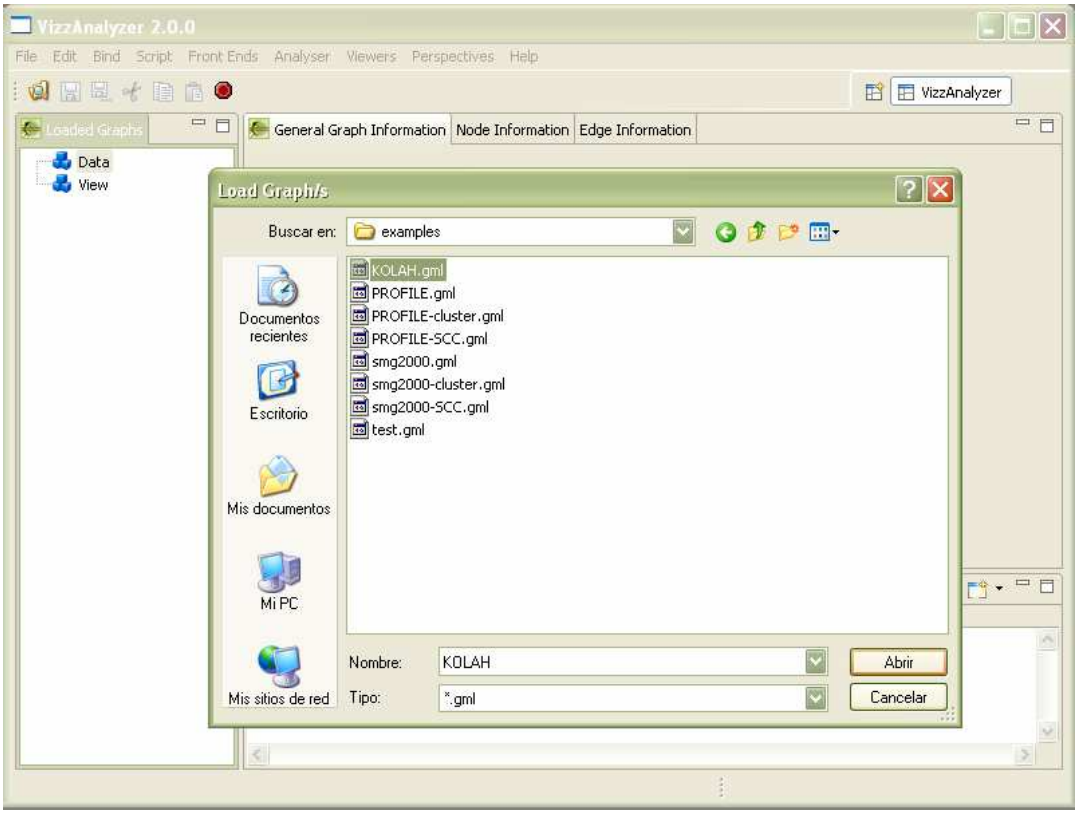

Figure A.5: Loading graphs dialog

The user can load another graphs, see information from them (node and edge information), or execute some actions over the graphs (copy, paste, cut).

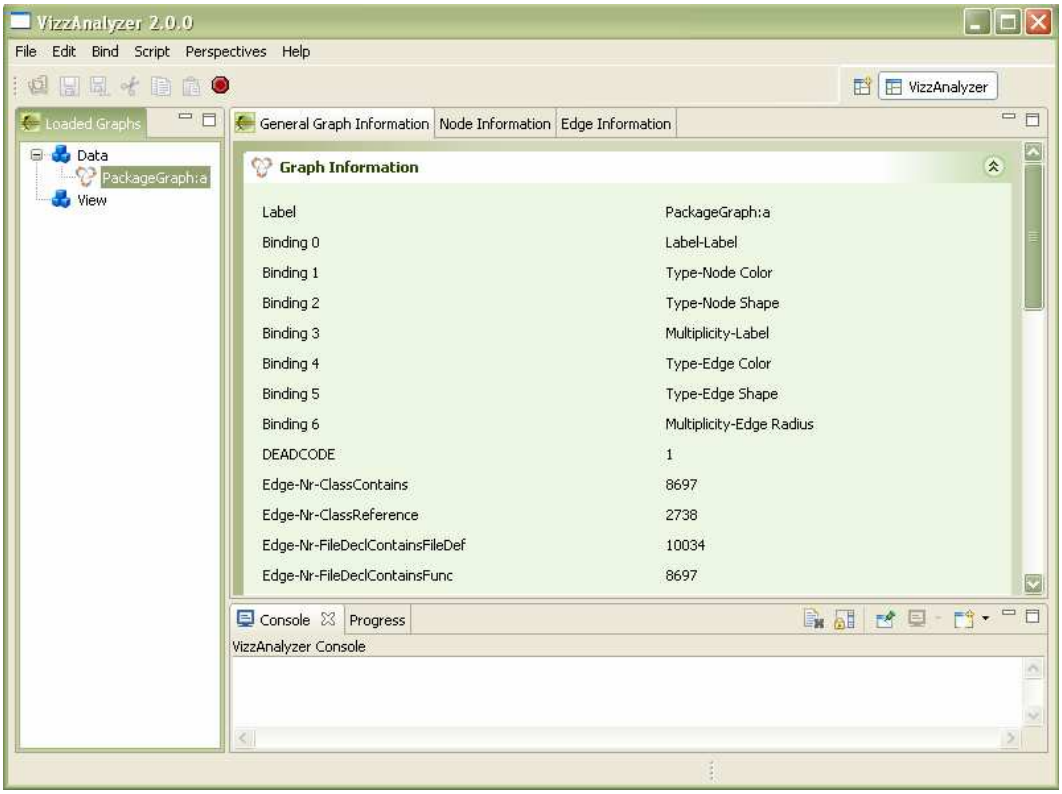

Figure A.6: Graphs loaded

Once a graph is loaded its information can be viewed, e.g., the General Graph Information as seen in Figure A.6: Graphs loaded. To provide a better overview the information for the nodes and edges is located in other tabs, as it is depicted in Figure A.7: Tabs information where it is possible to see the table for the Node Information. This is one of the improvements over the old VizzAnalyzer, where all views where in one and the same perspective.

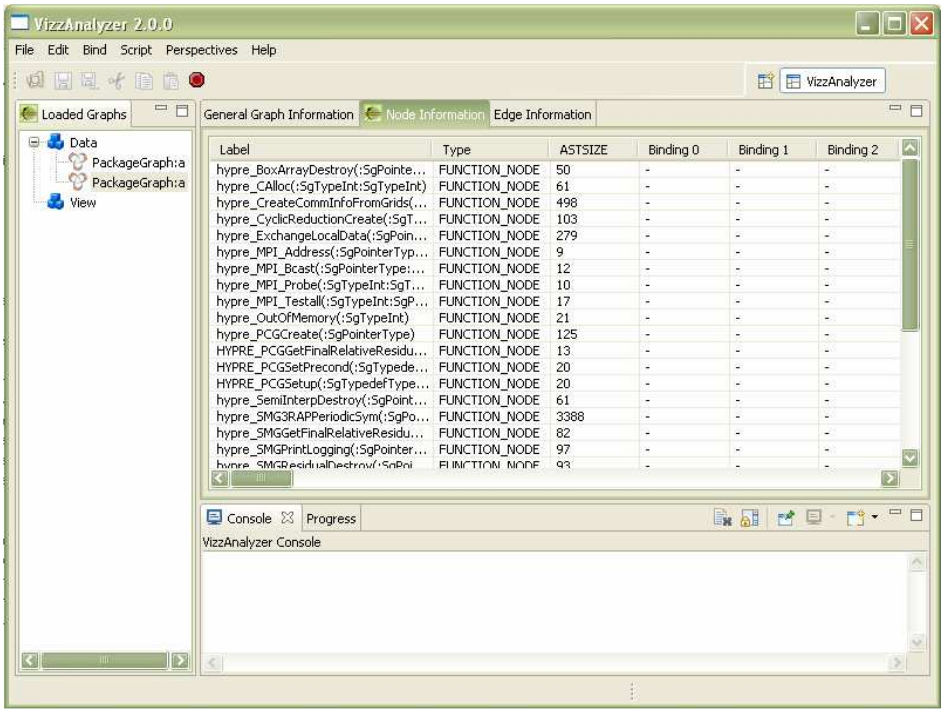

Figure A.7: Tabs information

Figure A.8: Perspectives depicts another view created for the VizzAnalyzer 2.0. In this case, all the information is on the same view. It is possible to switch between perspectives and all tabs designed can be extended or contract as the user likes.

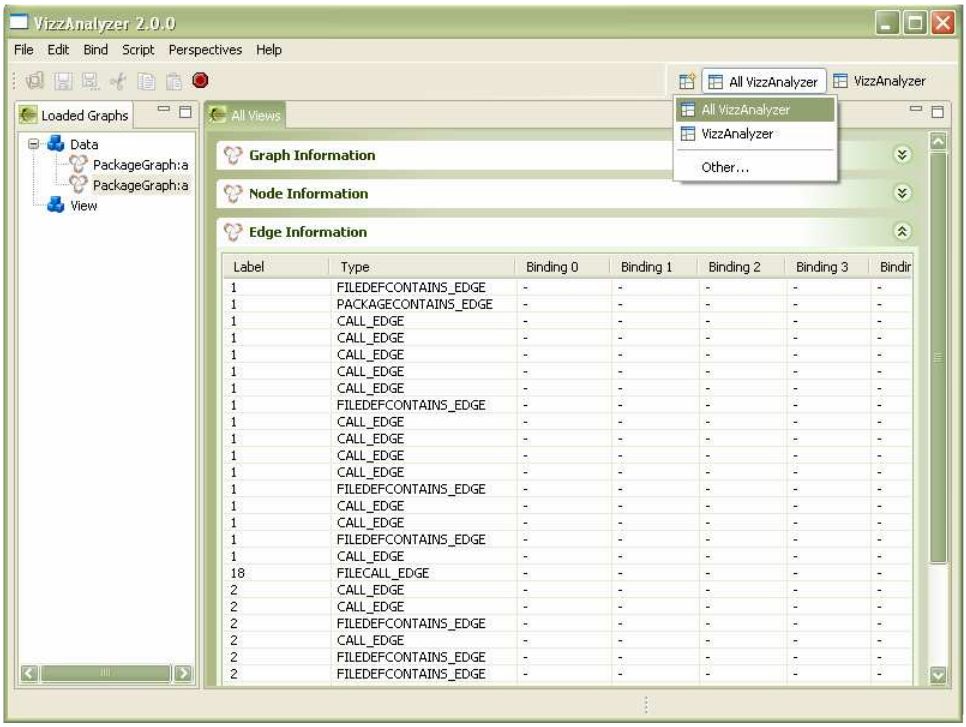

Figure A.8: Perspectives

A.1.4 Help manager

The help manager is intended to offer help and additional information about the features of the VizzAnalyzer and its plug-ins. The option is available from the Help menu bar as is shown in Figure A.9: Help manager menu.

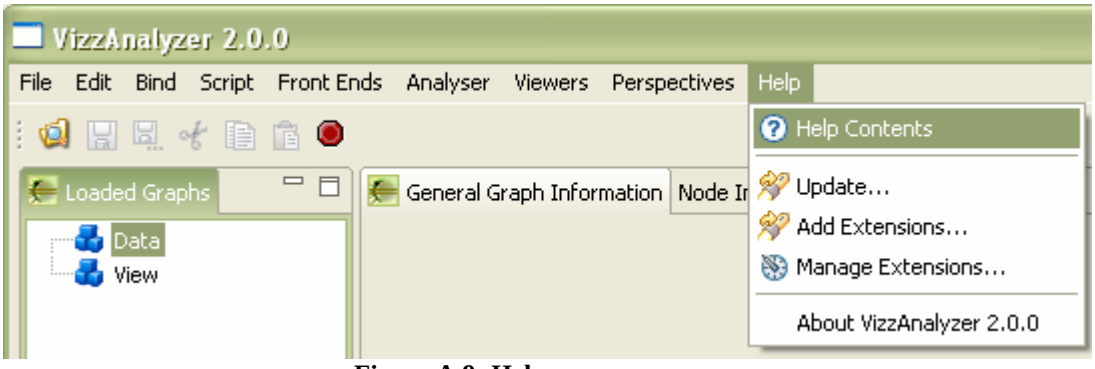

Figure A.9: Help manager menu

When the user presses *Help Contents* the available help contents are shown. In the current version, no information has been published, since it was not the focus of the thesis to generate help content. A.2.1 Updating help content in the appendix, explains for developers how to create new and update existing and publish help content for the system. Figure A.10: Help manager content depicts the standard view when the user asks for help.

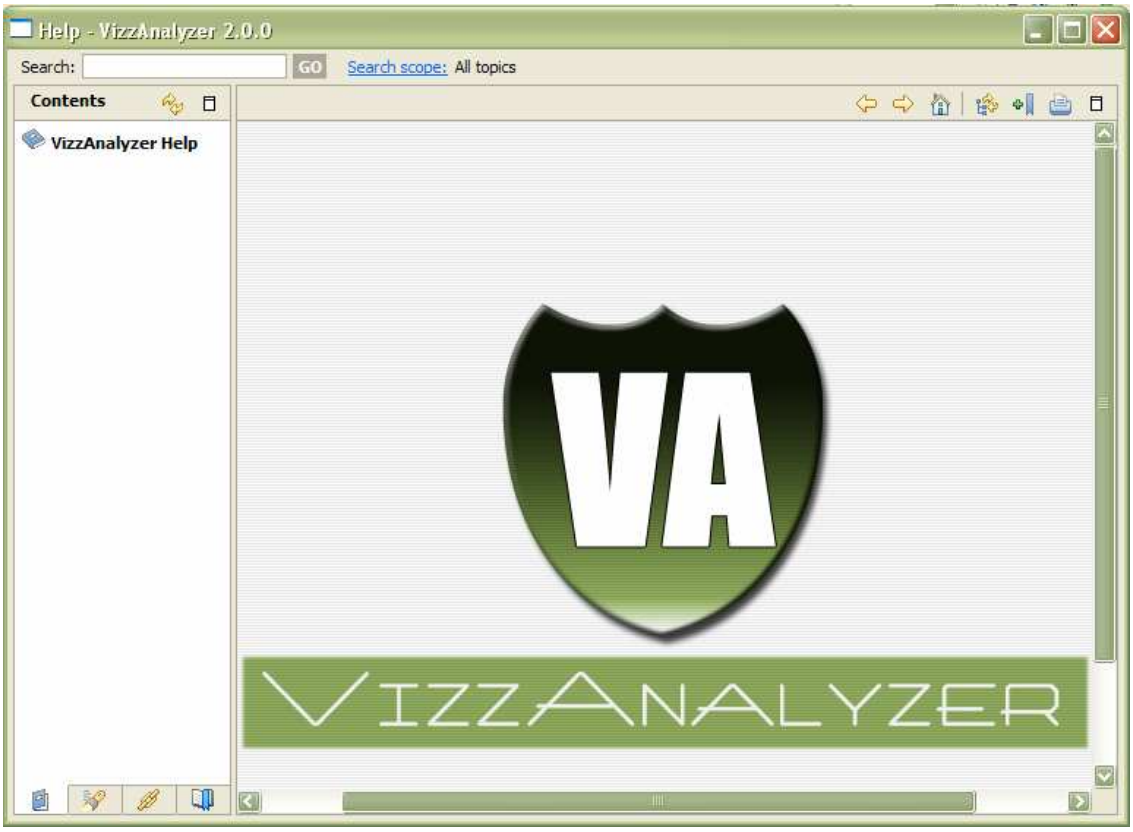

Figure A.10: Help manager content

The user can either browse the help contents manually, or he can perform a search on the content, which will be provided as Search results, as seen in Figure A.11: Help manager query. The left side shows a list of results for the query. When the user selects on one of the results, the right side shows all the content provided.

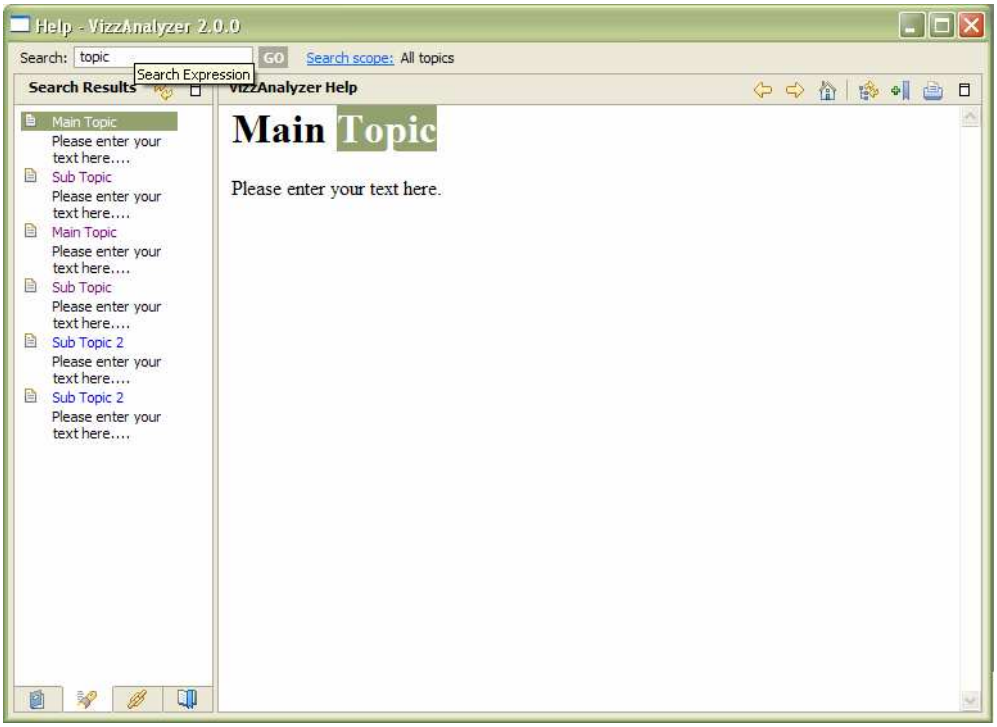

Figure A.11: Help manager query

A.1.5 Update manager

The update manager is available from the Help menu as well. It allows the user to check if a updated VizzAnalyzer version is available on the web. Figure A.12: Update manager menu shows how to start the update manager. The *Add Extensions* check if new features not yet implemented are available and the *Manage Extensions* lets the user enable, disable or remove already installed features.

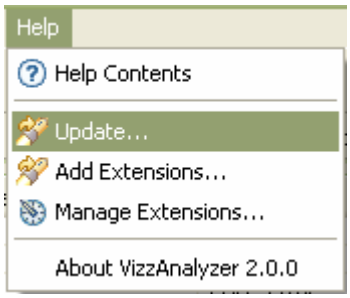

Figure A.12: Update manager menu

When the user check if a new version is available a progress bar as is showed in the status bar, refer to Figure A.13: Update progress. This is important because gives a feedback to the user about on the progress of the update action, which can take some time, depending on the quality of the internet connection.

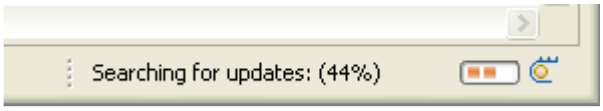

Figure A.13: Update progress

When the system is connection to the web page, a progress service informs the user about the progress. The user can wait until the operation is over, or he can choose to run the task in the background, while he is continuing his work. If the user does not want to wait, he can *Cancel* the operation. The progress service can offer more information about the operation which is available pressing the *Details>>* button. Figure A.14: Checking updates depicts how the progress service looks for the update manager.

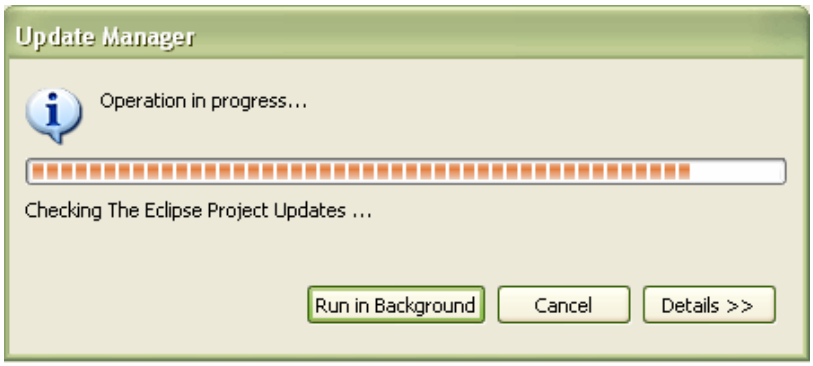

Figure A.14: Checking updates

The update manager checks the versions and provides the user with the search results as seen in Figure A.15: Taking new version. There the user can check new version to update to. The installation starts after a license agreement needs to be

accepted by the user and. After the installation is completed the system asks for re-start the application since it is necessary for the new version.

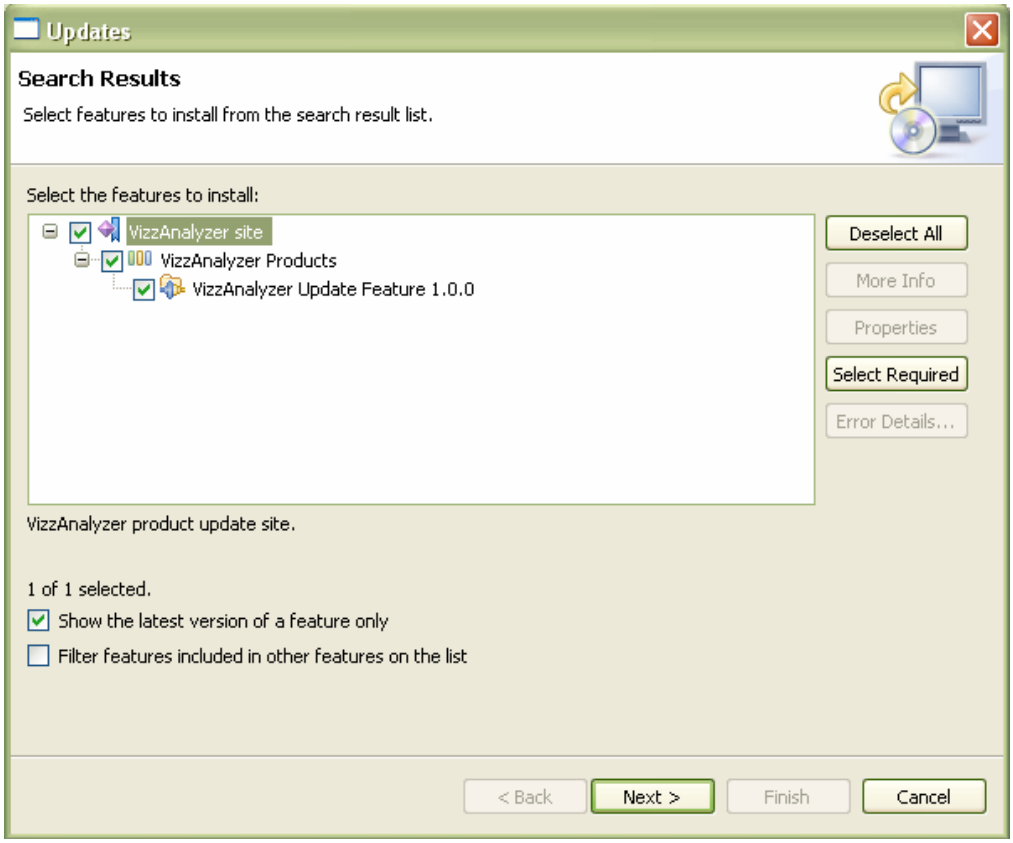

Figure A.15: Taking new version

A.2 Developer Manual

This chapter is intended to assist developers in using the new features introduced in the VizzAnalyzer. It will explain how to provide and update help content, how the update site manager works, how new builds of the VizzAnalyzer are exported, and how to the console and progress service.

A.2.1 Updating help content

The Help extension template has added three things to the VizzAnalyzer plug-in: a help extension, a XML table of contents (toc.xml) (2 and 3) and HTML stub files for each help page. The toc.xml file (2) defines a set of bookmarks for the help Web site. It allows freely setting up whatever structure best suits ones needs. It is possible to have as many categories with any content. Figure A.16: Updating help content shows a possible structure.

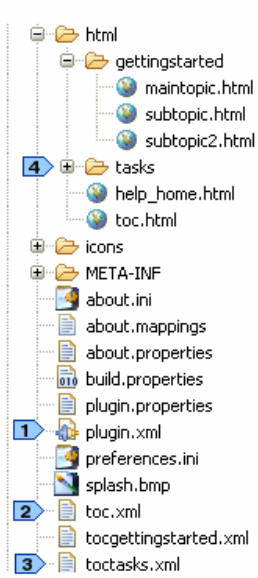

Figure A.16: Updating help content

The toc.xml as seen in Figure A.17: toc.xml defines an *anchor* for each category. *Anchors* are locations to which other toc files link adding their substructure. Notice how toc.xml defines the “tasks” anchor and “toctasks.xml” link to that anchor (Figure A.18:toctasks.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>
<toc label="VizzAnalyzer Help" topic="html/toc.html">
  <topic label="Getting Started">
    <anchor id="gettingstarted"/>
  </topic>
  <topic label="Tasks">
    <anchor id="tasks"/>
  </topic>
</toc>
```

Figure A.17: toc.xml

From Figure A.18:toctasks.xml, it is possible infer on the template. The HTML fields are just stubs, but they represent the real help content shown in the right pane of the help window. It is up to the content provider to fill in and link together the relevant content.

```
<?xml version="1.0" encoding="UTF-8"?>
<?NLS TYPE="org.eclipse.help.toc"?>
<toc label="Tasks" link_to="toc.xml#tasks">
      <topic label="Main Topic"  href="html/tasks/maintopic.html">
          <topic label="Sub Topic" href="html/tasks/subtopic.html" />
      </topic>
      <topic label="Main Topic 2">
         <topic label="Sub Topic 2" href="html/tasks/subtopic2.html"/>
      </topic>
</toc>
```

Figure A.18:toctasks.xml

A.2.2 Update site manager

In subchapter 5.7 the term *feature* was introduced. Features exist in two major places: **installed** (in the Eclipse product) and **packaged** (on the server as part of the update site).

Imagine that VizzAnalyzer is empty, no plug-ins and features. To get the functionality into it, it is necessary to download features from the server. *Features* must be published on a HTTP server in the following way:

Features and plug-ins are packaged for easy transportation over the network and all of them must be package into JAR archives. *Features* are placed into the *feature* directory and *plug-ins* into the *plug-ins* directory.

Eclipse provides a default implementation of the required interfaces that makes feature publishing easier and programming free. An *update site* has been created for testing when a new version has been released. Figure A.19: Update site created depicts the structure for the current *site*.

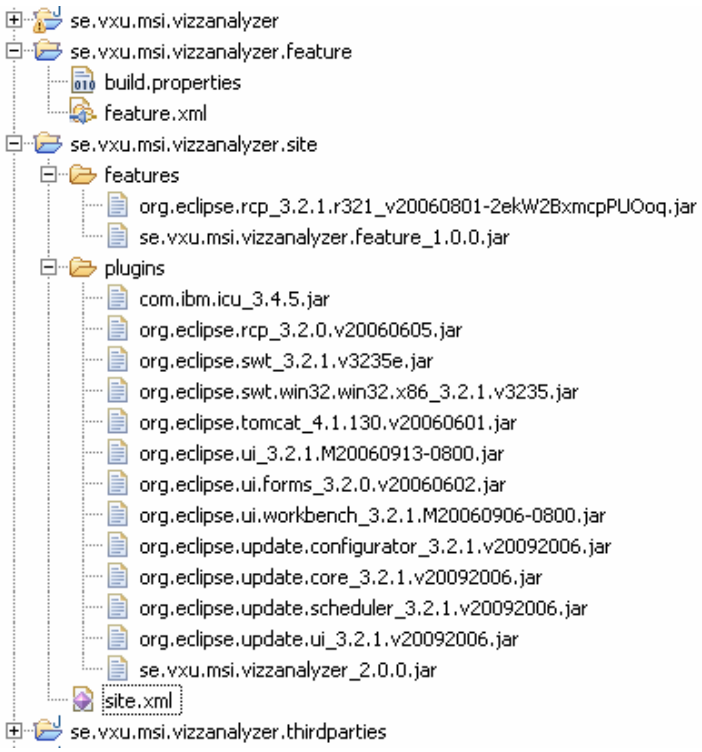

Figure A.19: Update site created

The first step is to connect the features in the workspace with the site. The update manifest editor is a combination of an editor and a builder which keeps track of the features in the workspace and builds them in a format required by the Update Manager. Figure A.20: Site editor shows the editor.

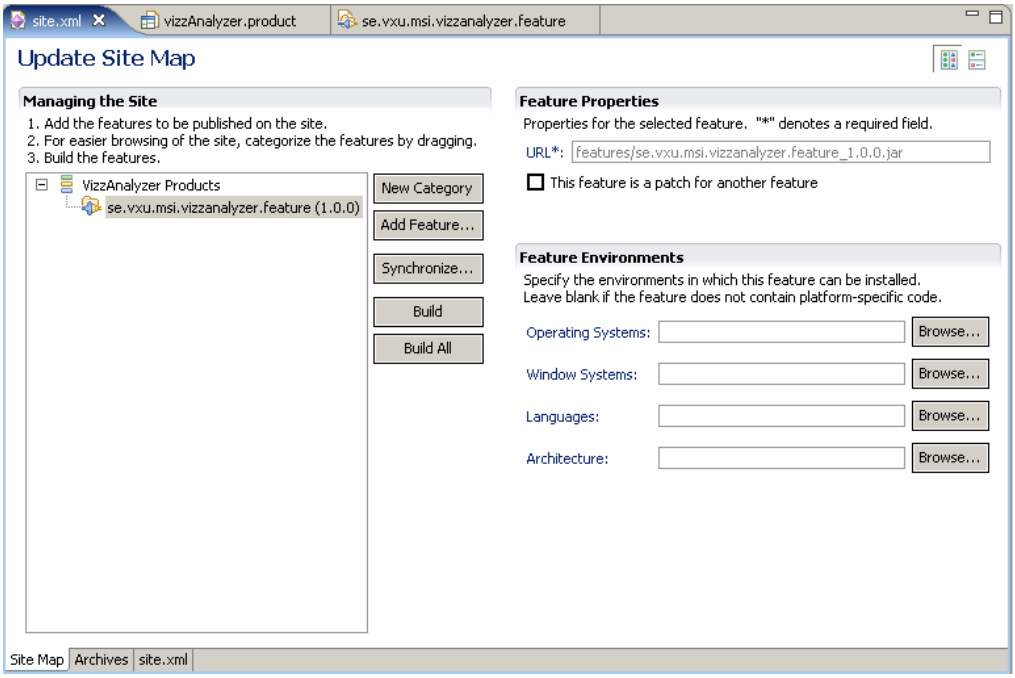

Figure A.20: Site editor

The packaged files are placed directly into the site project in the required folders. To connect features and the site use the **Add Feature…** button. After adding the features, it is possible to view the features on the site map. The site description is ready to be built. Click on **Build All**, this starts the export of the features to the site. In our example, only the VizzAnalyzer feature is built.

When the build is done, the site has *features* and *plugins* directories, as seen in Figure A.19: Update site created. All the plug-ins needed by a VizzAnalyzer feature are included on the site. Building an update site ensures that everything needed by the features is present on the site.

For testing purposes it is possible to use a local directory and to define the URL as “file:D:/eclipse3.2.1./workspace/vizzanalyzer.site/”, otherwise if there is a real URL just introduces its URL (see Figure A.21: URL site).

The site editor detects if the code comprising the *feature* has been modified, and rebuilds and republishes that feature, so that it is available for download to the end-user.

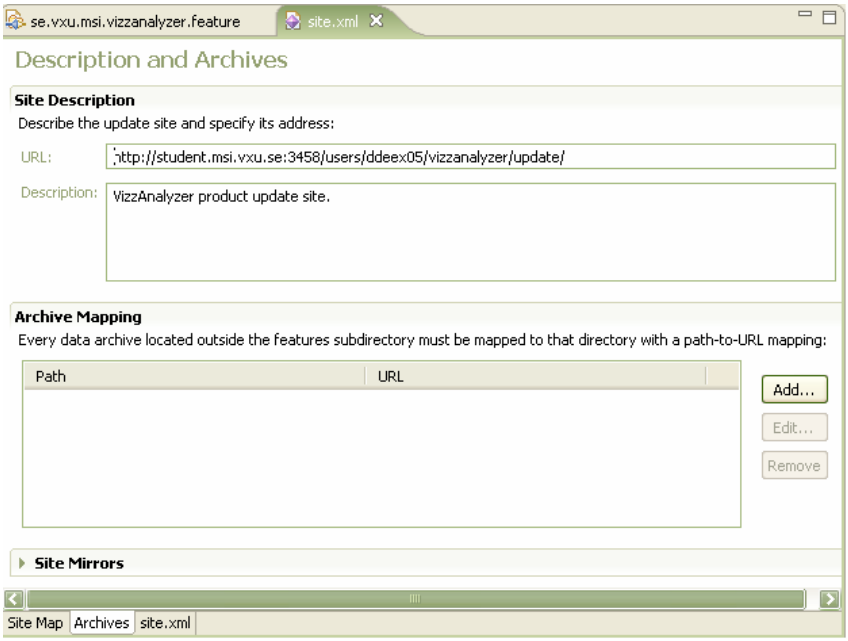

Figure A.21: URL site

A.2.3 Exporting VizzAnalyzer

VizzAnalyzer can be executed on the most commercial platform including Windows, Linux, Unix and MacOS. To do so the operating system independent core of the Eclipse platform is needs, as well as a set of operating system dependent plug-ins which are included in the *eclipse-RCP-3.2.1-delta-pack.* The delta pack contains all parts of the Eclipse SDK that are platform-specific. So, if you are on Windows and want to export VizzAnalyzer for Linux/GTK, you need the delta pack.

If the delta pack is not included in the current RCP distribution, it is necessary to download it, and to reload the configuration for the target platform.

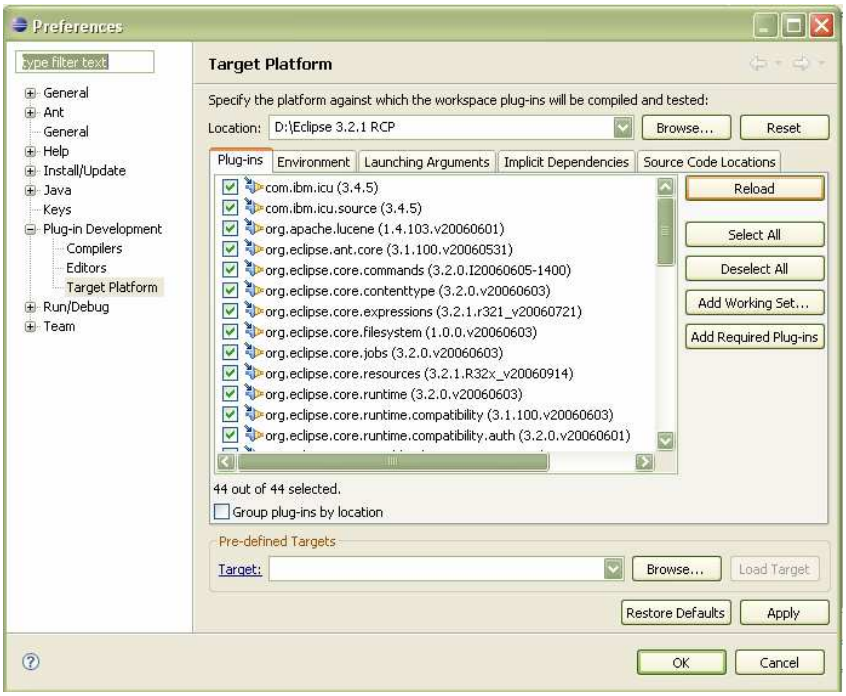

Figure A.22: Configuring delta pack plug-ins

The platform, for which the VizzAnalyzer shall be build, is configured in the product editor, which is part of the Export Wizard. Figure A.23: Exporting to other platforms has checked the *Export for multiple platforms* option, to build and export the VizzAnalyzer for more than one platform.

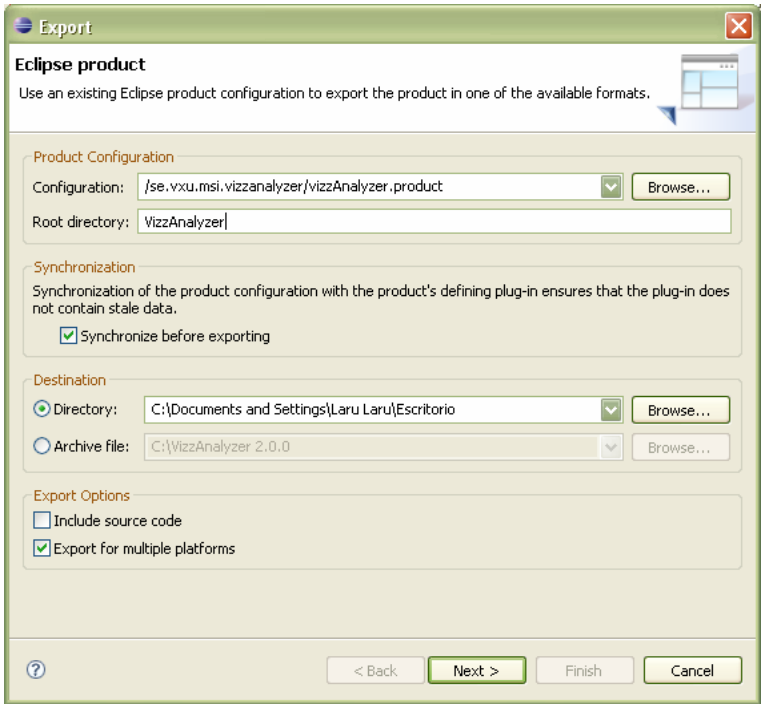

Figure A.23: Exporting to other platforms

In the next step of the wizard, depicted in Figure A.24: Select platforms to export, it is possible to select the set of platforms for which the VizzAnalyzer shall be exported.

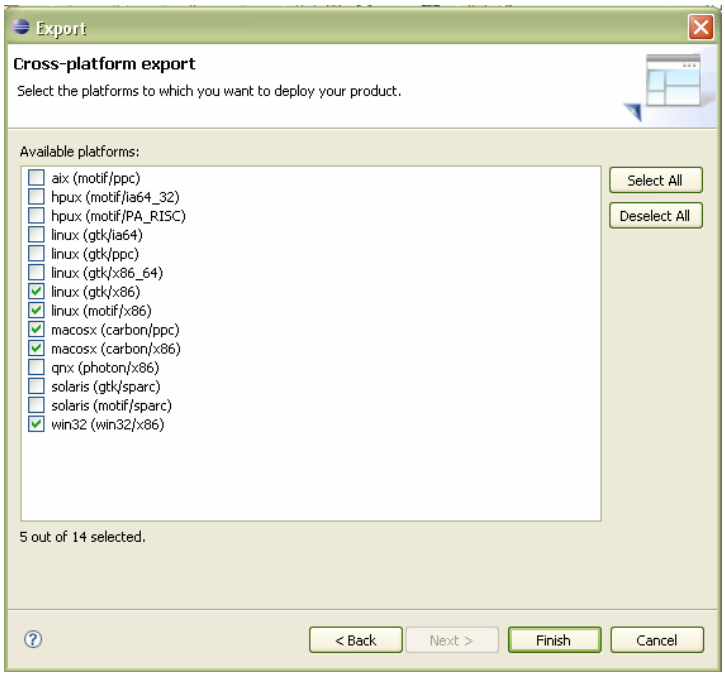

Figure A.24: Select platforms to export

The wizard generates a version for the VizzAnalyzer for each of the selected platforms. They are stored in separate folders, as it can be seen in Figure A.25: VizzAnalyzer OS.

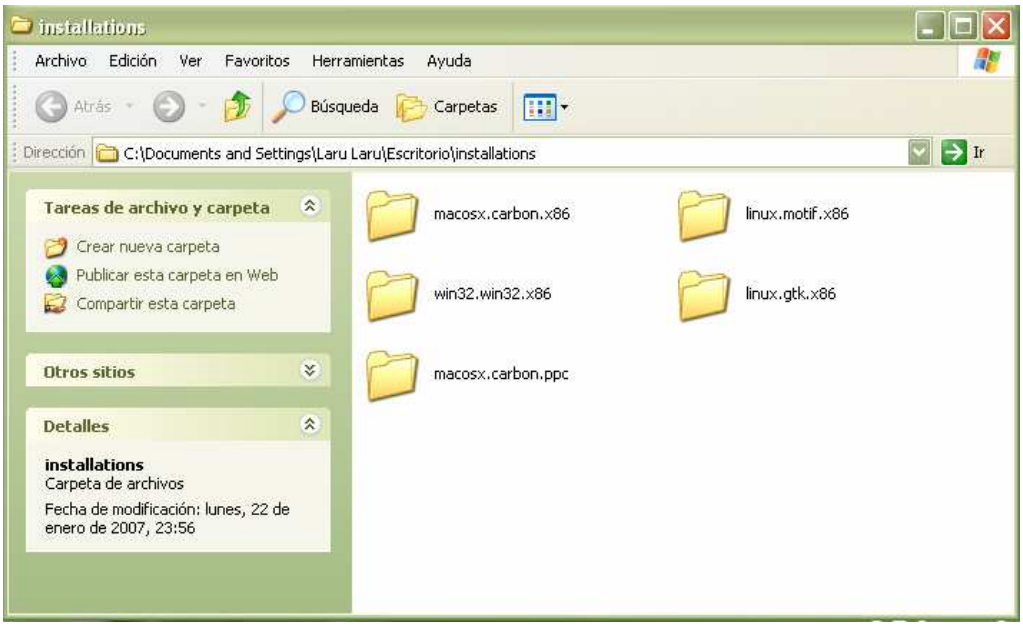

Figure A.25: VizzAnalyzer OS

A.2.4 How to use the VizzAnalyzer console

Since the System console has been replaced by a specialized console integrating better into the VizzAnalyzer. Current and future plug-ins and components need to write their output to this specialized console. This is possible using the code shown in Figure A.26: How to use the console, which does the following:

ConsoleTab.ID identifies the VizzAnalyzer console.

Any message wants to use. “Hello Console” has been used as a sample.

The VizzAnalyzer console defines different type of messages, and some of them are for information, error or warning. The console offers different types of messages as MSG_INFORMATION, MSG_ERROR and MSG_WARNING.

```
/** Code used from any class **/
1 ConsoleTab.println(ConsoleTab.ID,
2                    "Hello Console",
3                    ConsoleTab.MSG_INFORMATION);
```

Figure A.26: How to use the console

A.2.5 How to use the progress service

Since Eclipse version 3.0 provides a workbench a so called *progress service,* which combines the different progress services from SWT and previous Eclipse versions making their use easier and more centralized.

The VizzAnalyzer 1.0 provided a premature *progress service* which was not available from all parts of the application, and it had problems with canceling running processes. Eclipse provides different ways to implement a more advanced *progress service* for any application.

Figure A.27: Progress service busy indicator sample depicts how to implement a busy indicator. The user is not allowed to stop the process. It is just used to give a

feedback to the user. It is usually implemented for operations that need a short period of time.

```
      IWorkbench wb = PlatformUI.getWorkbench();
      IProgressService ps = wb.getProgressService();
1     ps.busyCursorWhile(new IRunnableWithProgress() {
        public void run(IProgressMonitor pm) {
2           ... try {
3                 monitor.beginTask("Performing decathlon: ", 10);
                  monitor.subTask("hammer throw");
                  //perform the hammer throw
4                 monitor.worked(1);
                  //... repeat for remaining nine events
                  } finally {
                    monitor.done();
                  }
        }
      });
```

Figure A.27: Progress service busy indicator sample

ps is a reference to the progress service and it is provided using the *IProgressService* from JFace and the *busyCursorWhile* method is implemented simply passing the *IRunnableWithProgress* instance.

Any method can be called to perform an operation. When the operation is executed and while the operation is running a progress bar is provided to the user.

BeginTask() method must be called which specify a description of the operation and the number of units of work that it will take.

Worked method specifying how many units spent. The *done* method is executed when the monitor finish the operation.

Figure A.28: Progress service job sample shows how to implement a so called Job. Jobs were introduced in Eclipse 3.0 and are operations that can run in background. While a busy indicator is not allowed to be cancelled a job is allowed to be cancel. The following is happening:

A *Job* called “Online Reservation” name is instantiated.

The *run* method has as a parameter a *monitor*. The monitor can be used to track whether the user wants to cancel the running Job. The running tasks have to check for this condition frequently, and abort its operation once the user requests to cancel.

A *job* can work both, in fore- or background, and it is possible to set the default. As shown in the example below, passing true to job.setUser starts the job as a foreground job.

The job is started using the schedule() method. This will start the job, after other background jobs have been completed.

```
1   Job job = new Job("Online Reservation") {
      protected IStatus run(IProgressMonitor monitor) {
            for (int i = 0; i < 1000000; i++) {
                  System.out.println("i--> " + i);
2                 if (monitor.isCanceled())
                        return Status.CANCEL_STATUS;
            }
            return Status.OK_STATUS;
      }
    };
```

```
3 job.setUser(true);
4 job.schedule();
}
```

Figure A.28: Progress service job sample

The progress service is itself a complex feature, which needs to be well understood to get the most out of it. For further information refer to [11].

A.2.6 Class path Configuration

This subchapter summarizes different articles about how Classpath works in Eclipse Project (see [30][31][32][33][34][35]).

Developers coming from a traditional Java programming are often confused by Classpath issues in Eclipse. A typical Java application has a global namespace made up of the contents of the JARs on a single universal Classpath. This Classpath is typically specified either with a command line argument to the VM or by an operating system environment variable. In Eclipse, each plug-ins has its own unique Classpath. This Classpath contains the following:

The OSGI parent class loader. All class loaders in OSGi have a common parent class loader. By default, this is set to be the Java boot class loader. The boot loader only knows about "rt.jar", but the boot Classpath can be augmented with a command line argument to the VM.

The exported libraries of all imported plug-ins. If imported plug-ins export their imports, you get access to their exported libraries, too. Plug-in libraries, import and exports are all specified in the plugin.xml. file." [30].

A JAR can be added to the Classpath of a plug-in different ways.

The JAR can be added to the boot Classpath. This is generally a bad idea, however, as it requires an extra VM argument, and it also affects the Classpath of all other installed plug-ins. If the JAR adds types –classes or interfaces – that mask types declared in other plug-ins, you will probably break those other plug-ins. Nonetheless, if you are looking for a quick and dirty hack, this is the easiest approach.

The JAR can be added to the declared libraries for a plug-in. This is fine if you don't anticipate a need for other plug-ins also to use that JAR.

A new plug-in can be created that is a wrapper for the library; then the new plug-in is added to the list of required plug-ins for all plug-ins that want access to the library.

Boot. The Java boot class loader. This is default OSGi parent loader, and has access to all JARs on the VM's boot Classpath.

App. The Java application class loader. This class loader has access to the traditional Classpath entries specified by the *–classpath* command line argument. In Eclipse this typically includes only the bootstrap class in *startup.jar*. The parent of the application class loader is the extension class loader.*"* [34].

Since the new VizzAnalyzer is based on Eclipse plug-in architecture it is possible to connect the plug-in to other plug-ins. Most plug-ins make use of services provided by other plug-ins or offer services that other plug-ins can consume.

The VizzAnalyzer 1.0 reads different plug-ins from a classpath command line and loads them to the menu.

For the current VizzAnalyzer 2.0, the approach should be different; the plug-ins already implemented and the new plug-ins developed for VizzAnalyzer should be based on Eclipse plug-in too, assuring the best cohesion between them.

Plug-ins define their interactions with other plug-ins in a number of ways. First, a plug-in can specify what other plug-ins it *requires*, those that it absolutely cannot live without. For instance, a UI plug-in will probably require the SWT plug-in. Plug-in requirements are specified in the *plug-in manifest file* (plugin.xml). The following example shows a plug-in that requires only the SWT plug-ins:

```
<requires>
    <import plugin=”org.eclipse.swt” />
</requires>
```

Figure A.29: Plug-ins required

At the same time, a plug-in can choose which classes and interfaces it wants to expose to other plug-ins. The plug-in manifest must declare what libraries (JARs) it provides and, optionally, what classes it wants other plug-ins to be able to reference. The following example declares a simple JAR file and exposes classes only in packages starting with the prefix *com.xyz.**.

```
<runtime>
  <library name=”sample.jar”>
    <export name=”com.xyz.*” />
  </library>
</runtime>
```

Figure A.30: Sharing JARs

Since there were some problems for the classpath when Vizzanalyzer 2.0 needs to read the dynamic plug-ins, I believe the best approach should be as follows:

Firstly, all VizzAnalyzer plug-ins from Analysis, Retrieval and Visualization plug-ins should extend from Eclipse platform too, so they can specify which the JARs files they share are.

Secondly, the VizzAnalyzer 2.0 could specify which (Analysis, Retrieval or Visualization) plug-in wants to interact. The interaction would be better since the own VizzAnalyzer and VizzAnalyzer plug-ins would follow the Eclipse architecture.

Figure A.31: VizzAnalyzer plug-ins required depicts how could interact the Analysis, Retrieval and Visualization plug-ins with. Each of these plug-ins could specify which are the classes that are shared, for instance, Figure A.32: Visualization Plug-ins share depicts, where Visualization shares only the classes specify on the package se.vxu.msi.vizzanalyzer.main.*.

```
<requires>
    <import plugin=”se.vxu.msi.vizzanalyzer.analysis” />
    <import plugin=”se.vxu.msi.vizzanalyzer.retrieval” />
    <import plugin=”se.vxu.msi.vizzanalyzer.visualization” />
</requires>
```

Figure A.31: VizzAnalyzer plug-ins required

```
<runtime>
  <library name=” visualization.jar”>
    <export name=”se.vxu.msi.vizzanalyzer.main.*” />
  </library>
</runtime>
```

Figure A.32: Visualization Plug-ins share

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470-70 80 00, fax +46(0)470-840 04
http://www.vxu.se/msi